6-30-2005

Linux OS emulator and an application binary loader for a high performance microarchitecture simulator

Scott Warner

Linux OS Emulator and an Application Binary Loader for a High Performance Microarchitecture Simulator

by

Scott Charles Warner

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science in Computer Engineering

Supervised by

Dr. Greg Semeraro
Department of Computer Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, NY
June 2005

Approved By:

Dr. Greg Semeraro
Primary Advisor – R.I.T. Dept. of Computer Engineering

Dr. Roy Czernikowski
Secondary Advisor – R.I.T. Dept. of Computer Engineering

Dr. Juan Carlos Cockburn
Secondary Advisor – R.I.T. Dept. of Computer Engineering

Thesis Release Permission Form

Rochester Institute of Technology
Kate Gleason College of Engineering

Title: Linux OS Emulator and an Application Binary Loader for a High Performance Microarchitecture Simulator

I, Scott Charles Warner, hereby grant permission to the Wallace Memorial Library to reproduce my thesis in whole or part.

Scott Charles Warner

Date

Dedication

To my wife Jodie and son Tyler whose support and encouragement made this possible. Thank you also for understanding all of the late nights and weekends spend at work while completing this thesis.

Acknowledgements

I would like to thank several people that have helped me through this process. The first person I'd like to thank is my advisor, Dr. Greg Semeraro, for his guidance and assistance throughout my thesis work. He was an invaluable resource throughout this process, providing the concept of RITSim and the components required to realize it, which provided the topic for this work, and the motivation, support and guidance to see it through.

I would like to thank Dr Roy Czernikowski and Dr. Juan Carlos Cockburn for their participation on my reading committee. Their input has helped to improve the quality of this thesis document.

I would also like to thank Mr. Paul Mezzanini for his assistance configuring the Linux machines to support aspects of the User Mode Linux work.

Finally I'd like to thank Ramesh Nagarajan and Theresa Bui, from Xerox, for providing the support and flexible work schedule needed to accomplish this while working full time.

Abstract

Simulation is a critical step in the development of state of the art microprocessors. Accurate simulation allows designers to confidently investigate various designs, while fast simulation times allow designers to thoroughly explore a design space. RITSim is an endeavor to create a high accuracy, high quality mircoarchitecture simulation infrastructure. This simulation infrastructure will be available for academic research in low power and high performance computer systems.

The scope of this work is to provide a Linux OS Emulator, a Binary Application Loader, and a Linux kernel running in a virtual environment for the RITSim project. In order to evaluate standard software loads and benchmark suites on target microarchitectures simulators must provide support for operating system calls. This may be accomplished with various levels of accuracy. Many past simulators chose to sacrifice simulation accuracy to improve simulation time, while others sacrificed portability and execution time for high accuracy results. This work provides three key elements to the RITSim environment in an effort to create a simulation environment that seamlessly combines both approaches to provide a single integrated tool that allows researchers to choose the approach that is best suited to their needs.

A first order simulation mode is provided that makes use of emulated system calls that are executed on the host computer's operating system to provide quick simulation times. This mode also maintains a high level of portability since the host operating system is used to access the hardware. A high accuracy mode is also available that runs in a highly detailed simulated operating system. When running in the high accuracy mode the simulated operating system must be loaded into a virtual environment allowing the

actual instructions of the operating system code to be simulated. Another key element is the binary application loader. This is required by the simulator to load executables into the simulator's virtual memory space and to prepare it for execution. This involves not only mapping or copying the executable into simulated virtual memory, but also the creation and initialization of a new user mode stack and configuration of the simulated processor's user mode registers.

Table of Contents

List of Figures

Glossary

RITSim	The Microachitecture, Power, Energy and Performance Simulator developed at the Rochester Institute of Technology.
DEC	Digital Equipment Corporation.
GPL	The General Public License or the GNU General Public License.
TLB	Translation Lookup Buffer.
UML	User Mode Linux.
LSE	The Liberty Simulation Environment.
ISA	The Industry Standard Architecture Bus.
x86	This is a reference to the 386, 486, 586, etc. family of Intel processors.
ELF	The Executable and Linking Format.
a.out	The Assembler OUTput format.
COFF	The Common Object File Format.
ECOFF	The Extended Common Object File Format.
GCC	The GNU C and C++ compiler.
GNU	A recursive acronym for “GNU’s Not Unix” .
bss	Defines a storage area in the data segment that holds un-initialized data.
rc.local	A script provided in the Linux kernel boot structure that is used to define custom initialization instructions.
PISA	Pendulum Instruction Set Architecture

Chapter 1 Introduction

RITSim is an endeavor to create a high accuracy, high quality mircoarchitecture simulation infrastructure. The intended use for this simulation infrastructure is for use in academic research in low power and high performance computer systems. To be effective, modern microarchitecure research requires a simulator that is accurate enough to show the effects of minor design changes. This is driven by the difficulty in obtaining performance gains in microachitectures. In an effort to keep advancing performance gains researchers are required to use either exotic solutions or a complementary minor design changes. To achieve this researchers need a simulation tool that will correctly simulate the effects of these minor changes. To drive this research the RITSim environment will provide an integrated simulation environment that combines accurate system simulation with an accurate microarchitectural simulation. This combination provides highly accurate simulation results giving researchers the information needed to study the impact of minor design changes.

One of the unique aspects of RITSim is that it combines a quick, first order simulator, with a highly accurate, full system simulator seamlessly integrated within the same environment. This work will cover three key elements of this environment. The first component is an emulated operating system. This is provided to meet the quick first order simulator goal. Operating system emulator based simulators are an ideal way to provide quick simulation times. The reason for this is that the simulator actually executes the system calls on the host machine instead of simulating them. The drawback to this approach is reduced accuracy. Since these system calls are executed outside of the simulation environment the simulator does not incorporate the effects of the operating system on the application. This is acceptable though for a high-speed first order simulation. To give researchers the freedom to run a large set of applications in the simulator the Linux [16] OS emulator provides emulation support for nearly all of the Linux system calls. Past simulators provided limited system call support, limiting the number of programs that may be executed within that environment. Providing the support needed to execute any application is very important as researchers attempt to understand the impact of modern software designs on microarchtecure.

The next aspect of this work adds the support of a high accuracy, full system simulation mode. To obtain the best simulation results, the simulation environment must support a full system simulation. This requires an operating system running in a virtual environment. Since the operating system runs in a virtual environment, the operating system calls fall within the scope of the simulation environment.

The final feature is a binary application loader that is used to load executables into simulated memory space. This feature is required to load the application into simulated virtual memory space and prepare it for execution by the target processor.

Chapter 2 Thesis Objectives

2.1. Motivation and Goals

Computer architecture researchers rely heavily on simulations to drive new microprocessor designs. The time and monetary costs associated with the construction of a hardware prototype are simply too prohibitive to justify without first evaluating new ideas with thorough simulation [9]. Simulation tools also play an important role in research work where the end goal may not be to create a physical microprocessor. These tools allow researchers to probe new ideas to increase their knowledge of microprocessors and explore advanced ideas and designs.

There are two main types of simulators. Simulators such as SimpleScaler [5] and RSIM [12] provide an emulated operating system where system calls are converted from the simulated call to a call that is executed on the host. While offering quick execution times and a high level of portability these simulators suffer in terms of accuracy. This is due to the fact that system calls are executed on the host computer not inside the simulation environment thereby overlooking the importance of the operating system [13]. The result of this approach may lead to inaccurate or misleading simulation results, especially for operating system intensive operations. Consider database applications that rely heavily on the scheduling of multiple processes and file system accesses both of which are the responsibility of the operating system. An investigation of database transaction activity revealed that close to 40% of its execution time is spent in the operating system [10]. Web server applications represent another set of operating system intensive applications. These applications make extensive use of the network services, file system accesses, and significant scheduling load to service multiple requests all of which are provided by the operating system. An experiment that ran a Webstone benchmark test on a Zeus server reported more than 70% of the execution time occurred in the operating system [10]. This disconnect with the operating system may lead the researcher to draw inaccurate conclusions. The danger is that an interesting new design may be overlooked. The opposing case would be the prototyping of a promising new design, only to find that in was a substandard design during physical testing.

Clearly high accuracy simulation results are a key goal for a new simulation tool. High levels of accuracy are offered by simulation tools such as SimOS [10] and L_RSIM [11]. These tools offer substantially more accurate results by simulating the operation system. While increasing accuracy these approaches also dramatically increase the simulation time. Often processor models are simplified to alleviate the simulation time penalty. The result is a simulation environment that produces accurate results for operating system intensive applications, but fails to simulate the intricate interactions with a modern microarchitecture [11]. An additional drawback is the lack of portability. This type of simulation environment is created for a specific target architecture, operating system and simulation environment. These effects contribute to code that is not portable. An example would be attempts to extend the popular SimOS [10]. SimOS [10] was developed at Stanford to simulate the Flash [17] multiprocessor. Attempts have been made to extend this simulator to support Alpha [18], PowerPC [19], x86 [20] and SPARC [21] architectures. Only two of these ports were successfully completed. The Alpha [18] port was completed by Western Research Laboratory (formerly Digital Equipment Corporation, DEC). The Alpha [18] architecture has similarities with the MIPS [22] architecture, which the Flash [17] architecture was based on, helping to reduce the complexity of the Alpha [18] extension. The Austin Research Lab completed the PowerPC [19] port. It is important to note that large research labs completed the two successful ports. Attempts to extend the simulator to support the x86 [20] and the SPARC [21] architecture have failed [23]. Clearly the tightly coupled target architecture, operating system and simulation environment contribute to create an environment that severely impacts portability and code reuse.

The Microarchitectual, Power, Energy and Performance Simulator (RITSim) project will improve upon prior simulator approaches to provide a single environment that provides a highly accurate microarchitecture simulator for low-power and high performance microprocessors that will be available to the academic environment. Simulating the operating system in SystemC will provide high accuracy. A parallel simulation environment will be utilized to increase the performance of the simulated operating system. Inclusion of an emulated operating system will allow quick first order simulations as well as maintain a high level of portability.

The topic of this thesis is to provide the emulated operating system, binary loader and virtual environment from which the simulated operating system is run within the scope of RITSim. Linux is a widely used operating system available under the General Public License (GPL) that runs on many of today’s microprocessors, making it an excellent choice for academic use. The GPL allows for free distribution and access to source code, both of these are vitally important within the scope of this project. Another benefit is that it’s a standard operating system widely used in the academic environment alleviating the need to learn a new operating system when using this simulator.

There are two parts to the Linux OS Emulation tool. The first is to provide a mechanism to handle intercepted system calls. In Linux, system calls are issued to the Kernel by the User Mode processes to access hardware devices. This provides an abstraction layer between the user and the hardware that the operating system is running on [1]. This is typically referred to as a system call by proxy method. When the user program executes a system call the simulator traps it and sends it to a system call proxy where the call is converted to a system call that is executed by the host operating system. When the host system finishes executing the system call, the results are repacked and sent back to the simulator. The goal for the interface to this proxy is to provide an approach that is extensible, allowing for future expansion. An example of this may involve using base classes, with derived classes that contain the pertinent data structures.

The binary loader will be responsible for allocating the necessary system memory, transferring the binary files to that memory, and setting any necessary instruction pointers or other registers required to allow the code to execute. There are many applications that need to load Linux binary executables. Selection of the best mechanism for our use involves studying what other applications, such as User Mode Linux, do to provide this service.

A virtual environment will be provided to load and execute the simulated operating system. An existing software package, User Mode Linux (UML), provides a good starting point for this feature. UML creates a virtual Linux machine that runs on top of the host machines Linux kernel. This has been used very successfully in the past for testing new Linux Kernels [8]. Furthermore the UML kernel code is executed within the simulation environment so the effects of the system calls are seen within the simulator. For example, cache and TLB pollution effects due to the kernel code and the effects of context switching would be seen and the impact of these events would be measurable in the simulator. Other avenues were explored prior to making the final decision. Existing simulators that feature simulated operating systems, such as L-RSIM [11] and SimOS [10], were studied in an effort to select the best approach.

The operating system that was selected was Linux. Linux is a very popular operating system in the academic environment due in large part to the fact that is a freely distributed, open source operating system. The clear advantage of this is the low acquistion cost, but perhaps the most important aspect is that it gives developers access to the source code. This gives researchers the freedom to modify parts of the operating system to fit their needs. An example would be modification of the existing scheduling routine to optimize if for real-time operations. Another benefit for RITSim is the large number of existing software applications available for Linux. This gives the researcher the ability to run a wide range of programs on the target architecture instead of limiting it to a few test suites. Other simulation environments such as L-RSIM [11] used a custom operating system. This approach simplifies the simulation environment since only the functionally needed by the simulator is provided. The two main drawbacks to this approach are the unfamiliarity of new developers with the custom operating system and the poor correlation between the simulation operating system and the operating system used on an actual machine.

2.2. Supporting work

The majority of recent papers published have relied on the simulation results achieved from the freely distributed simulation tools that include SimpleScaler [5], Rsim [12], and SimOS [10]. Each of these simulators supports either system call emulation or simulated operating systems.

SimpleScaler [5], in development since 1994, is a widely used simulation tool in academia. SimpleScaler is an architectural, execution-driven simulator. Architectural simulators are a category of simulators that implement the high level architecture, as opposed to simulators that implement the highly detailed, low-level microarchitecture. Architectural simulators are generally faster and more portable, but offer less accurate simulation results. Execution-driven simulators are simulators that execute the simulation program and generate the simulation stream dynamically. The complement to execution-driven simulator is trace-driven simulation, which reads the trace of an instruction saved from the previous execution. Trace-driven simulations are simpler to write since there are no functional components and no feedback from the trace. Perhaps the biggest drawback to trace-driven simulation is the loss of accuracy because they are not able to provide feedback on speculative results that are supported by superscaler processors [15].

SimpleScaler falls under the category of simulators that emulate system calls. A proxy handler intercepts system calls coming from the simulator, converts them to a system call that can be executed on the host processor, executes the call and copies the results of the call back to the simulated program's memory [5]. It is here that the disconnect with the operating system is realized, since all system calls are handled outside the simulator. This approach has its advantages however, two of which are execution time and portability. Since the system calls are executed by the host hardware, instead of simulated hardware they execute much quicker. Portability is also maintained by relying on the host processor’s system calls; the simulator is unconcerned with the underlying hardware of the system. This approach is not completely free of portability issues though, moving this simulator from one operating system to another will require modification of the system call translator to correctly map to the host operating system’s system calls. Another drawback to this particular implementation is the lack of support for all system calls in Linux. This limits the number of applications that can be executed in this simulator. The Linux OS Emulator will provide the option to use emulated system calls similar to SimpleScaler to maintain the benefit of quick execution time and portability. In addition to this all Linux system calls will be emulated, this will allow the simulator to execute any Linux program.

SimpleScaler [5] includes different simulation programs that provide increasing levels of simulation detail. At the fast execution end are sim-fast and sim-safe. These are optimized for simulation speed and cannot be used to perform microarchitectural simulations. These simulation programs simply execute instructions on a simulated machine with no architectural features. Essentially these simulators simply verify that the instruction set simulator is working correctly. It may also be used by individuals who want to add instructions to the machine, modify a compiler and verify that the program still functions correctly. Sim-fast is the quickest, while sim-safe adds some memory operation safe guards. This makes sim-safe useful for debugging sim-fast. Sim-EIO is also a fast executing simulation similar to sim-fast and sim-safe, but adds external trace and generator capabilities. Sim-profile falls somewhere in the middle; the main benefit of this simulator is that it profiles by symbol and address. It provides reports on many simulation results including dynamic instruction count, instruction class counts, usage of address modes, etc. The sim-cache/sim-bpred models provide fast simulations for cache miss and miss-prediction rates, however it offers no timing impact. Sim-outorder is the most detailed simulator that supports an out-of-order execution core, speculative execution, a 2-level cache and branch prediction. Of course being the most detailed it also has the slowest execution time. A source of confusion when using the different simulations is that the simulation results rarely agree. For example, cache statistics from sim-cache differ from the same statistics generated by sim-outorder bringing into question the accuracy of either set of results.

Another execution-driven simulator RSIM [12] from Rice University was created to research shared memory multiprocessors that exploit instruction-level parallelism (ILP). Developed in 1997 this simulator improved on contemporary simulators, such as SimpleScaler, by providing a more detailed processor model. Simpler processor models may be used to increase simulation performance, again at the cost of accuracy. These simpler models did not do an adequate job of simulating ILP processors [12]. Under some conditions RSIM provides very accurate results, however this simulator does not simulate the operating system. Once again operating system intensive workloads will not be accurately simulated.

SimOS [10] was developed to address the inaccuracies associated with neglecting the impact of operating system activities on simulation accuracy. By providing full-system simulation capabilities simulation accuracy is dramatically increased for operating system intensive applications. Another benefit to simulating the entire machine is that it is capable of booting and running a fully functional operating system and any application that runs on that operating system. The downside to this approach is that simulating the entire system takes a lot of time. This was addressed by providing three levels of simulation speed through the use of simulation models that vary in the level of detail. While being a valuable simulation tool it has its limitations. The processor and cache models are simplified limiting the simulated accuracy of the complex interactions of modern microarchitectures [11]. Due to its lack of support for complicated processor and cache models, SimOS is best used as a system level simulation environment, as opposed to a processor architecture simulation environment. RITSim will offer detailed processor and cache models with the added benefit of a simulated operating system making it a simulation tool that is much more useful for simulating modern processor architectures.

L-RSIM [11] was a further refinement to existing simulation tools. In an effort to create a simulation environment with high accuracy models and operating system simulation support it combined the highly detailed processor model of RSIM with a simulated operating system and simulated I/O device behavior. To simplify the task of simulating an operating system a hybrid operating system, LAMIX [11], was created. This operating system was specifically targeted at file system and disk I/O operations thereby simplifying the implementation when compared to simulating a complete existing operating system such as Linux. This simulator showed an excellent correlation between simulated and actual results obtained from a SGI Octane workstation for architectural performance parameters such as memory performance and disk seek times [11]. The operating system performance, however, did not correlate well indicating that while using a non-standard operating system simplified the simulator development it also negatively affected simulation results. This simulation tool provides an indication of the highly accurate simulation results that are possible when a simulated operating system is coupled with highly detailed processor and cache models. This work will take this one step further by supporting a standard operating system to improve simulation accuracy.

The Liberty Simulation Environment (LSE) [4] represents the most recent simulation tool in the academic environment. This simulation environment addresses the disconnect between hardware mapping and the software models used to simulate them. It provides a set of resources that automatically generate software models based on the hardware description, thereby maintaining the accuracy of the model. To allow these models to be automatically generated a set of well-defined interfaces were defined for the communication between models. One benefit of the consistent communication protocol between models is the increased portability. It also creates a set of flexible reusable components that are relatively easy to reconfigure to support different architectures. LSE is not is a simulator, but an environment that is used to create a simulator whose goal is solve the hardware mapping to software model consistency issue, which falls outside the scope of this work.

The Linux OS Emulation tool within the scope of RITSim will provide a single environment that incorporates many of the desirable features of past simulation tools. This is accomplished by using a standard operating system with an emulation mode that will provide fast simulation times and provide portability coupled with a simulated operating system that provides increased simulation accuracy for operating system intensive applications.

Chapter 3 Background on Existing Academic Solutions

3.1. System Call Emulation Based Simulators

While there are many simulation environments currently available, one of the most widely used simulators is SimpleScaler [5]. This chapter begins with a detailed description of how the system calls are simulated, and then moves on to look at some other relevant simulators.

3.1.1 The SimpleScaler Environment

The following is a detailed description of how SimpleScaler [5] handles simulated system calls. A system call proxy is used to convert simulated (target machine) system calls to system calls that are executed on the host machine. System calls are emulated by the following sequence:

1. Decode the system call. This is represented by an enumeration that corresponds to a system call.
2. Copy system call inputs from target memory to host memory prior to execution on the host. The amount of memory varies by system call. For example, in Linux, the `sys_read()` call will need to allocate an area the size of the file that is to be read in. Additionally the input variables are copied to the correct registers for the host machine. Again this varies by system call, in Linux the `sys_exit()` call only takes one input variable while the `sys_select()` call takes six input variables.
3. Execute the system call on the host machine.

4. Copy the system call results from host memory to target memory. For example if `sys_read()` was executed the results need to be copied from the host buffer into simulator memory space.

5. Set target result register to indicate successful completion or the error status of the system call. Set any other registers that are affected by the result of the system call.

The `sys_syscall()` function is the call used to execute an emulated system call via the system call proxy. It is defined as follows:

```
void sys_syscall(struct regs_t *regs,  /* registers to access */
                 mem_access_fn mem_fn, /* generic memory accessor */
                 struct mem_t *mem,    /* memory space to access */
                 md_inst_t inst,       /* system call inst */
                 int traceable);       /* traceable system call? */
```

The `regs_t` structure represents the registers that are used for this system call. This structure is shown below:

```
struct regs_t {
     md_gpr_t regs_R;            /* (signed) integer register file */
     md_fpr_t regs_F;            /* floating point register file */
     md_ctrl_t regs_C;           /* control register file */
     md_addr_t regs_PC;          /* program counter */
     md_addr_t regs_NPC;         /* next-cycle program counter */
   };
```

The `mem_access_fn` accessor is used to access the simulated virtual memory space. It is defined below:

```
typedef enum md_fault_type
    (*mem_access_fn)
        (struct mem_t *mem,  /* memory space to access */
         enum mem_cmd cmd,   /* Read or Write */
         md_addr_t addr,     /* target memory address to access */
         void *p,            /* where to copy to/from */
         int nbytes);        /* transfer length in bytes */
```

The `mem_t` structure represents the memory space that is accessed. This structure is defined below:

```
struct mem_t {
  /* memory object state */
  char *name;             /* name of this memory space */
  struct mem_pte_t *ptab[MEM_PTAB_SIZE];/* inverted page table */

  /* memory object stats */
  counter_t page_count;   /* total number of pages allocated */
  counter_t ptab_misses;  /* total first level page tbl misses */
  counter_t ptab_accesses; /* total page table accesses */
};
```

The `md_inst_t` value is a little misleading, this value does represent a system call number but it is used only for debug, the actual enumeration that represents the system call that will be executed is in the global variable that represents the `V0` register (for the Alpha architecture). This illustrates one of the difficulties associated with modifying SimpleScaler[5], the use of global variables to store the register values for the system calls makes it very difficult to trace the call through the code.

The design for this thesis work will make use of the basic sequence used by SimpleScaler [5]. This is a good proven approach, however, there is room for improvement. One area is the method used to map the target register set to the host system call parameters. With SimpleScaler[5] the registers used in the system call emulator are tied directly to a particular ISA. This makes it necessary to rewrite this code for each ISA, an example of this would be separate sets of source code to handle Alpha and PISA architectures. For this work a layer of abstraction will be added to enable the use of a single system call emulator for various ISAs, the instruction set simulator changes, but the system call emulation should not. An additional improvement is the emulation of all of the system calls, which gives a researcher the freedom to execute a wide range of programs, instead of limiting them to a particular test suite. Other design details of the SimpleScaler environment that will be avoided are the use of global variables and macros. The use of these significantly degrades readability and reuse, both of which are important goals for this work. Finally, the interface used to copy data between the simulator and host memory will be simplified.

3.1.2 SimpleScaler Derived Simulators

Being a widely used simulation environment that is freely distributed other simulators such as The Simulator for Multithreaded Computer Architecture, SIMCA [24], and SIMCORE [25] have made use of SimpleScaler [5] to varying degrees. There are two approaches to creating a simulation tool, modifying one that exists, or creating one from scratch. The benefit of modifying an existing simulation tool is the time saved by not having to create the entire environment from scratch. The drawback is the difficulty involved in modifying the existing tools without introducing errors. Many existing simulation tools, such as SimpleScaler [5], whether through the extensive use of global variables and/or macros, are difficult to extend. Many academic simulation tools did not start with the goal of providing a flexible, modular environment. In order to modify such code without introducing errors one must fully understand its internal details, which is often a nontrivial task due to difficulties in understanding someone else's code and limited documentation [26]. One of the benefits of creating a new simulation environment is that is allows the developer to draw from past implementations to create the best possible environment for the target application. The main drawback to this approach is the time needed to create all of the tools. This section examines two simulators, one that added a layer on top of SimpleScaler [5] to achieve the desired simulation support, and another that rewrote an improved simulation environment based on SimpleScaler [5].

SIMCA [24] was developed at the University of Minnesota; it is a simulator targeted at simulating super-threaded architectures. In an effort to speed up the development time of the simulation environment the simulator was implemented on top of a SimpleScaler Release 2.0 [5] simulator. This approach used a technique called process-pipelining to place a layer on top of the existing SimpleScaler simulator. It allowed them to extended SimpleScaler to support multithreaded simulations. This approach was used in an effort to minimize development time by leveraging existing software. The penalty for this approach though was that simulations were 3.8 to 4.9 times slower [26]. While this shows that it is possible to make use of existing simulators to speed development time, the resulting simulation environment will not operate at optimal speeds.

SIMCORE [25] is a simulation environment developed at The University of Electro-Communications in Tokyo, Japan. The primary intent of this simulator was to provide a tool for academic education and research. The simulation functionality of SIMCORE [25] is similar to the sim-fast simulation in the SimpleScaler toolset [5]. This simulation is optimized for fast simulation speeds and enhanced readability compared to SimpleScaler [25]. This makes it a great tool for education since results may be obtained quickly. Use as a research tool is limited, however, since it is optimized for speed over high accuracy. It makes use of the sim-safe/sim-fast simulations from SimpleScaler [5], which cannot simulate the impact of architectural changes directly. In SimpleScaler[5] the architectural changes must be simulated using sim-outorder to verify the performance impact of those changes. Furthermore, examination of the system call emulator code revealed that the simulation environment only implements the system calls needed to run the SPEC CPU2000 test suite, further reducing its usefulness as a research tool.

SIMCORE [25] improves on SimpleScaler [5] by, among other things, removing global variables and macros to improving readability. A single class holds all of the information needed to perform an instruction. It was discovered, however, that while offering improvements in some areas, it still relies on SimpleScaler [5]. The binary application loader uses a modified version of the SimpleScaler [5] loader and verification is performed using SimpleScaler[5] sim-safe simulations.

Clearly it’s an unsuitable tool for in-depth research, but there are some good points about SIMCORE [25] that will parallel the approach used in this work. The first of which is the use of C++ to improve reuse through abstraction layers. The second is improved readability by removal of global variables and macros.

3.1.3 Other Simulators

A very popular x86 emulator is BOCHS [27]. It simulates an entire x86 based system allowing x86 software to be run without modification on any host machine. The main purpose of this tool is to allow a person to run an x86 operating system (Windows, Linux, etc.) and software on a non-native host system. Even though this emulator is not targeted at architectural research the system call emulation method was examined as an additional approach to emulating system calls. The approach that is used in this emulator is to create a virtual environment for the emulated operating system. As the emulated operating system makes calls to a particular hardware driver to perform an action on the hardware, the driver converts the emulated system call to the host system call and invokes it on the host hardware. An example of this would be a call to the CDROM driver to spin up the drive, the CDROM driver would convert the `ioctl` call used to spin up the drive to the host specific call and execute the converted system call on the host operating system, which in turn spins up the CDROM on the host machine. The way various host operating systems are supported is the use of `#ifdef` statements surrounding system calls to check for the host operating system. The use of all of the `#ifdef, #elif` statements leads to poor readability. However, the biggest drawback to this approach is the large number of driver files that must be modified to support new host operating systems. Every driver file must be modified wherever there is a system call to support the new system calls. This approach will not be used for RITSim.

As discussed in section 2.2 RSIM [12] is another popular simulation tool. RSIM [12] emulates some system calls, such as read and write, and simulates other system calls at the CPU level. The emulated system calls are relevant to this section and shall be discussed here. RSIM uses a class to trap the system calls; these calls are then passed to a proxy class that decodes the call and handles the emulation. The emulation is typical of other examples that have been examined, where the target system call is converted to execute on the host machine. The `write` system call, for example, copies the buffer data associated with the write command to a local buffer, extracts the register values associated with the emulated call converts them to the host system call parameters, makes the system call on the host machine, and returns the return status of the host based system call. The steps used to emulate the calls are similar to the ones that will be used for this work, but the design of the RSIM emulation does not readily support emulation of different architectures since it still relies on specific register mappings.

3.2. Binary Application Loader

The binary application loader section will examine the mechanisms present in Linux that are used to load an executable binary file and prepare it for execution. This will be followed by a look at the mechanisms used by SimpleScaler to load the simulated programs.

3.2.1 The Linux Binary Loader

In Linux the `execve()` function is used to replace the execution context of a process with the new context contained in an executable file [1]. Part of this process is loading the executable binary file and configuring the stack to prepare for execution. This is handled by the `load_binary()` function. Linux supports many executable formats, such as *Executable and Linking Format* (ELF), *Assembler OUTput Format* (a.out), MS-DOS EXE programs, BSD Unix's COFF executables, etc. To correctly interpret and load each of these different formats there are different versions of the `load_binary()` function, each tailored to the target executable. For example the `load_elf_binary()` function is called to load the ELF executable. For the purpose of this work only the ELF format will be supported. ELF was selected because it is the current standard Linux executable format. An additional benefit is that it is an extremely popular format for Unix systems [1].

The `load_binary()` function is responsible for loading the executable binary code and configuring the stack. Since RITSim will only use the ELF format the following analysis is based on the `load_elf_binary()` function. This method executes approximately twenty-one steps to load an application that was compiled using dynamic library linking. For RITSim the loader will only support executables that were compiled with statically linked libraries. One very important aspect of a simulation environment is producing consistent results. Achieving this goal requires the use of statically linked libraries. Since all of the libraries are included in the executable, the simulation may be rerun at any time with the knowledge that no libraries were modified from one run to the next. A benefit of this requirement is a significant decrease in the number of steps required to load the executable. A major simplification presented by loading statically linked libraries is that the loader does not need to search for the shared library and the interpreter needed to execute it. Keep in mind that a dynamically linked ELF file may link to executables of a different format. For instance an ELF executable may contain a dynamic link to an a.out library. To support this the loader must also load an interpreter for a.out executables. The following discussion describes the process used by the `load_elf()` method to load a statically linked ELF executable.

The `load_elf()` function reads the header of the executable file and performs consistency checks on the magic numbers. If the executable does not satisfy the loader's requirements an error of `-ENOEXEC` is returned.

If it is an acceptable executable file the program header information must be interpreted, these headers contain the information about the program segments and the shared library information. For RITSim the shared library information is an indication that the executable contains dynamically linked libraries, which is a failure case.

After the segment information has been obtained the `flush_old_exec()` function is called. This function removes all traces of the currently running executable. The `PF_FORKNOEXEC` flag is then set in the process descriptor. This flag is used to track processes and is set when a process is forked and cleared when it executes a new program.

This is followed by a call to `setup_arg_pages()`. This function allocates memory for the new process's user mode stack and inserts the memory region in the process's address space. It then assigns page frames containing the command line arguments and environment variable strings to this address space, so they can be copied to the new user mode stack.

The text and data segments are then mapped with the `do_mmap()` function. This creates a new memory region that the text segment and data segment of the executable are mapped to. The un-initialized data section (bss), which follows the data segment, is then mapped using the `do_brk()` function.

The `start_code`, `end_code`, `start_data`, `end_data`, `start_brk`, `brk` (although inconsistent the name is `brk`, not `end_brk`), `start_stack`, `env_start`, `env_end`, `arg_start` and `arg_end` fields of the process's current memory descriptor are then updated.

Finally, the `start_thread()` macro is called to modify the user mode register `eip` to point to the execution starting point and the user mode register `esp` to point to the top of the new User Mode stack. At this point the process is ready to begin execution. For the RITSim binary application loader the sequence should be similar. The exception being the use of simulated virtual memory as the target for loading the executable, containing the user mode stack, and simulated user mode registers.

3.2.2 The SimpleScaler Binary Loader

SimpleScaler [5] provides various routines to load executable files into simulated virtual memory space. There is support, via separate files, for the two main architectures supported by SimpleScaler[5], Alpha and PISA. Both of these implementations support binary loading using the Binary File Descriptor Library (BFD) or native Extended Common Object File Format (ECOFF) access methods. The BFD is a library that provides a single interface to read and write object files, executables, archive files, and core files in any format. The benefit of the BFD library is that it provides a set of methods to gain information about an executable through a common interface for different executable formats. There is a compile time flag that needs to be set in the makefile to direct the loader to use the BFD libraries, if not defined, the loader reverts to the native ECOFF format. While the individual commands used to gather information about the executable differ from those used for ELF executables, the overall process of loading the executable and preparing for execution is quite similar.

The purpose of the SimpleScaler[5] loader function is to load the program text and the initialized data into simulated virtual memory space and to configure the program segment range variables to prepare for execution. This process varies slightly depending on whether the BFD loader is enabled or if the file is loaded using the native ECOFF functions. If the BFD loader was enabled, the loader process begins by opening the executable file and checking for the correct file and endian format. If the formats are correct the file is read into memory. At this point all of the section headers are read, any section that is `allocated`, `loadable` and not `NULL` are read into a buffer. After all sections have been stored in the buffer it is copied into simulated virtual memory. If the section is `loadable` but not allowed to execute it is determined to be the bss section. To handle the bss section the loader creates a buffer of the bss size and zeros all of the values, then copies this to simulated virtual memory. With the sections stored in simulated virtual memory, the loader examines each section name to identify the text and data segments. Upon identification the header information is used to determine the location and size of each segment.

From this point on the implementation is the same for both the BFD loader and the native ECOFF loader. The final steps involve simple sanity checks on the text and data segments to make sure they were actually found in the executable. The stack pointer is then setup. The `agrc` value is copied onto the stack then the `argv` array pointer and the `argv` array followed by the `envp` pointer and the `envp` array. Finally the stack and instruction pointers are set for the target processor and the program counter register is set with the program entry point.

The portion that loads the executable using the native ECOFF methods begins by reading the executable into memory. This implementation differs from the BFD Loader by verifying the endian and executable format after the file is read into memory. The start and size of the text and data segments are read from the ECOFF header. When the text and data segments are located they are stored in a local buffer then copied into simulated virtual memory space. After the text and data segments are copied into simulated virtual memory space, the process continues as above with the initialization of stack and processor registers.

The SimpleScaler [5] binary loader process is similar to the Linux binary loader. The most significant difference is the need to copy the executable into simulated virtual memory space. This will be the same for the RITSim loader.

The general sequence used by both loaders is to evaluate the header to verify that the executable meets the loader's requirements. Then copy or map in the text, and data sections. Followed by copying the `envp, argv, argc` variables to the stack. Finally the processor registers must be set with the new stack pointer and instruction pointer. This is the same sequence that will be used on the RITSim binary application loader.

3.3. Simulated Operating System Based Simulators

Simulators that support a fully simulated operating system are necessary to obtain the most precise timing measurements. This is especially true when running simulations with applications that are operating system intensive, such as database or web-server applications. As discussed in the supporting work section SimOS [10] and L-RSIM [11] are two academic simulation environments that provide simulated operating systems. This section takes a closer look at these two approaches

3.3.1 The SimOS Simulation Environment

SimOS [10] attempts to simulate an entire machine. The simulation time associated with simulating an entire machine or system is extensive. To combat this SimOS provide three simulation levels. The first level is called the positioning mode, which provides the fastest simulation times with the least accuracy. It is a very useful mode to run to gain initial results quickly. The next level is called rough characterization mode and is a compromise between fast simulation speed and accurate simulation results and is a good intermediate step to narrow down a design. The final level is accurate mode, which provides the most accurate results at the expense of simulation times.

Different device models are used for the each of the simulation levels, however even in the most accurate mode the processor and cache models are very basic. For instance there are two processor models implemented for the accurate mode. The first is the Mipsy processor. This models a simple single-issue pipeline processor, which uses a straightforward fetch-decode-execute loop [10]. To speed up execution times this model simply charges a fixed latency for each instruction instead of modeling the processor pipeline. Thus limiting the use of this model for in-depth processor architecture investigations.

A second more detailed processor model is supplied. The MXS model uses the same fetch-decode-execute loop as Mipsy, but also offers more detailed modeling typical of a modern superscalar processor. While this model provides more detail, the data it generates is targeted at overall system level performance. This again limits its usefulness for detailed processor architecture investigations.

While targeted as a system simulation tool, SimOS has significant limitations when used as a processor simulation tool. In contrast, by providing a simulated operating system combined with detailed processor and cache models RITSim will provide a simulation environment that is conducive to in-depth processor architecture investigations.

3.3.2 The L-RSIM Simulation Environment

As discussed in the supporting work section, L-RSIM [11] is a simulation tool that is an extension to RSIM [12]. The purpose of this simulator was to provide increased simulation accuracy by adding a simulated operating system. This approach coupled the highly detailed ILP processor model from the RSIM [12] environment to a simulated operating system. The goal was to improve the accuracy of simulations that ran operating system intensive applications. While it provides a simulated operating system, the actual simulated operating system proves to be a weakness. To avoid the complexity of porting an existing operating system to the simulation environment a simplified operating system was used. This custom operating system was targeted at file system and disk I/O. Verification of this simulator demonstrated accuracy improvements over RSIM [12] for file and disk I/O interactions. There was, however, a poor correlation for operating system performance between the simulator and an SGI workstation that it was verified against.

Although the use of a simplified operating system simplified the simulator implementation and showed some good performance correlations, the best results would be obtained from simulating a standard operating system. Of importance is that this approach does demonstrate the benefit of incorporating a simulated operating system in the simulation environment. The next logical step would be the inclusion of a standard operating system within the simulation environment. This was addressed by the use of Linux as the simulated operating system provided by RITSim.

Due to the shortcomings of the existing solutions RITSim will make use of an existing application, User Mode Linux, to provide a Linux kernel running within a virtual environment. User Mode Linux is actually a patch that is applied to a Linux kernel. This patch to the kernel produces an application that loads a full Linux operating system into a virtual environment that is run on top of the Linux kernel on the host machine. The primary use for this is to test new kernel versions, debug new software, kernel experimentation, etc. User Mode Linux is ideal for this type of work since it is run in a virtual environment, where all of the resources are contained within a single file system on the disk. When running applications from within User Mode Linux nothing outside of this file may be touched, therefore eliminating the risk of damaging the kernel on the physical system. Some of the latest uses of this have involved testing network applications over complex networks all within a single physical machine. Eliminating the cost and time involved in configuring multiple machines to perform the same testing.

Within the scope of this work User Mode Linux will be used to provide a standard operating system, Linux, loaded into a virtual environment that provides the base for a fully simulated operating system.

Chapter 4 Design

4.1. *System Call Interface Design*

A new system call interface was designed drawing from previous implementations such as SimpleScaler [5], RSIM [12], and SIMCORE [24]. The goal was to create an efficient, readable and easily expandable interface. Linux system calls, when run on an x86 platform, may use up to six registers to store parameters. To execute a system call, the integer value representing a system call is placed in register `eax`, any other registers needed to execute the system call are then loaded with the correct value. When the required registers have been correctly set the system call is initiated by calling the software interrupt `0x80`. For example, the `sys_read()` call is run by programming the `eax` register with the system call index associated with the `sys_read()` call of `0x3`, `ebx` is programmed with the file descriptor, `ecx` is programmed with the character buffer pointer, and `edx` is programmed with the file size. Following the completion of the system call an integer value is passed back in `eax` that contains the completion code, and the character buffer has the data that was read in with the `sys_read()` call. In RITSim the target system call will be captured by the instruction set simulator. It is also responsible for capturing the register values for the target architecture and invoking the emulated system call method.

To support the emulated system calls a set of registers will be needed to store the required system call parameters. Simulators such as SimpleScaler [5] and SIMCORE [24] use generic register descriptors such as `r[0]`, `r[1]`, etc. This approach provides a small level of abstraction since it is independent of the actual register name that is associated with different architectures. This approach, however, does not allow the target architecture to change without extensive modification to handle the varying register sets used in differing architectures. Although the register names are abstract, the meaning of these registers are still hard coded to a specific architecture. For example, for the Alpha architecture, `r[16]` is the register that holds the file descriptor for the read operation. So the system call proxy extracts the value in `r[16]` to get the file descriptor, this is fine for an Alpha architecture, but any architecture that uses a different register to store this value will be incorrect. Clearly the system call proxy will need significant rework to function correctly with different architectures. This important aspect was addressed by providing a parent register class that is used by the system call proxy and child register classes that are used by the instruction set simulator to convert from the target register set to a generic register set that will be used by the system call proxy. The effect of this is an emulator that will run different architectures without modification. The instruction set simulator will correctly map the target registers to the register sets needed by the system call through accessor methods defined in the parent register class. The child class will provide the concrete implementations of the accessor methods to provide the correct mapping from the target architecture to the generic system call register set. This approach allows the developer to simply create a new child class for each architecture to handle the register mapping while the system call proxy only needs to reference the parent class, which eliminates the need to rewrite the system call proxy. This greatly simplifies this aspect of simulating different architectures since only the instruction set simulator needs to reference the child specific to that the target architecture, but the system call proxy remains unchanged. There are other significant benefits to this approach. One important benefit is the ability to utilize different ISAs without changing the underlying OS emulation, allowing RITSim to support ISA / compiler research. Another important benefit is that this mechanism also supports different existing ISAs easily. The instruction set simulator needs to change to support the target ISA, but the OS emulation portion will remain unchanged. These benefits make this a very flexible solution.

As illustrated by the read example the other major aspect that needs to be addressed is the memory space that the simulator uses. In the read example the result of that call was that data was read into host memory space. This needs to be moved into the simulator's memory space prior to returning from the simulated system call. The converse is true for write operations. Clearly some sort of memory descriptors are necessary to handle the movement of data from the simulator's memory space to host memory space prior to a system call or from host memory space to simulator memory space after the system call. Previous approaches allocate a new buffer in host memory space as needed for the operation prior to making the host system call; then various methods are called to either copy from simulator memory to host memory or host memory to simulator memory. The parameters needed to complete the copy are: the pointer to the buffer in simulator memory space, a pointer to the buffer in host memory space, and the size of the buffer. The simulator memory space pointer and buffer size should be supplied in the system call register variables. The host buffer is allocated as needed, so the pointer to that host memory buffer is readily accessible. The `MemoryUtility` class that will handle the movement of data between the simulator and host memory spaces will supply two methods. While the pointers are accessible from the registers the system still needs to know which direction the data is being copied, whether from host memory or from simulator memory. Thus the two copy methods will be required to specify the intended direction of the memory transfer. The supplied methods are:

```
data_t CopyFromSimulator(void* sourceAddr,
                         void* destinationAddr,
                         data_t sizeInBytes)

data_t CopyToSimulator(void* sourceAddr,
                       void* destinationAddr,
                       data_t sizeInBytes)
```

For this work the underlying code for the copies will simply be `memcpy()`. The reason for this is that at this point in the RITSim development everything is still in the same memory space so no conversions will be needed. As the simulation environment grows this will need to be expanded to handle referencing different memory spaces. Additionally, this could be tied into the cache policy to determine any actions that need to be taken by the cache to support the data movement providing highly accurate simulation results. For future expansion an integer value is returned from this method, initially the method will always return 0. As the memory class grows, however, this should be used to return failure modes.

Invoking emulated system calls will be handled with a single method that is passed a pointer that encapsulates the registers. The following call is used to invoke a system call:

```
void SystemCall (SysCallParameters *sysParam)
```

The method takes a pointer to the `SysCallParameters` class as the input argument. This class contains the register variables needed to perform the system call. It was named

`SysCallParameters` in case it needs to contain data other than just register values. The `SysCallParameters` are set using the following call:

```
      void SetTargetRegisters(data_t targetReg[])
```

Where the `targetReg` is an array of registers associated with the target architecture. This call copies the register values from the target register set to the generic register set used to call the host system call. This method should be called prior to executing the emulated system call. Upon returning from the emulated system call the following method is called:

```
      void GetTargetRegisters(data_t targetReg[])
```

This method copies the values of the generic register set to the target register set. This should be called following emulated system calls to copy the results of the calls back into the target register set. Again, this design allows for maximum flexibility in the future as the RITSim project grows to include multiple ISAs.

The following class diagram illustrates the classes associated with the system call emulation.

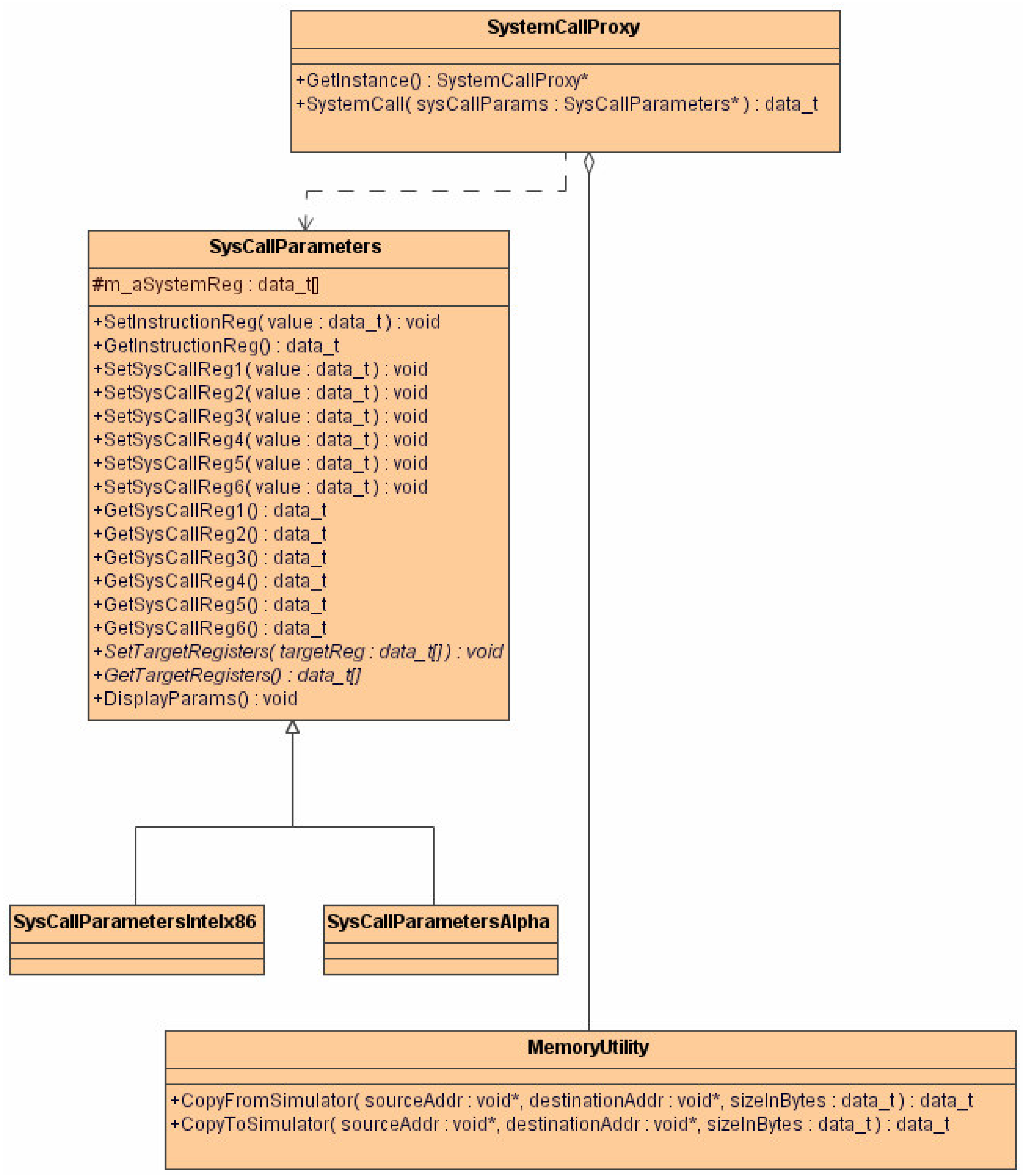
SystemCallProxy
+GetInstance() : SystemCallProxy*
+SystemCall(sysCallParams : SysCallParameters*) : data_t
SysCallParameters
#m_aSystemReg : data_t[]
+SetInstructionReg(value : data_t) : void
+GetInstructionReg() : data_t
+SetSysCallReg1(value : data_t) : void
+SetSysCallReg2(value : data_t) : void
+SetSysCallReg3(value : data_t) : void
+SetSysCallReg4(value : data_t) : void
+SetSysCallReg5(value : data_t) : void
+SetSysCallReg6(value : data_t) : void
+GetSysCallReg1() : data_t
+GetSysCallReg2() : data_t
+GetSysCallReg3() : data_t
+GetSysCallReg4() : data_t
+GetSysCallReg5() : data_t
+GetSysCallReg6() : data_t
+SetTargetRegisters(targetReg : data_t[]) : void
+GetTargetRegisters() : data_t[]
+DisplayParams() : void
SysCallParametersIntelx86
SysCallParametersAlpha
MemoryUtility
+CopyFromSimulator(sourceAddr : void*, destinationAddr : void*, sizeInBytes : data_t) : data_t
+CopyToSimulator(sourceAddr : void*, destinationAddr : void*, sizeInBytes : data_t) : data_t

Figure 1. System Call Proxy Class Diagram

The next class diagram shows the relationship of the instruction set simulator to the `SystemCallProxy` and the `SysCallParameters` classes. The intention is that the instruction set simulator will instantiate a child parameter class, say `SysCallParametersIntelx86`, to convert the target registers to the generic system call registers through the supplied conversion method. The instruction set simulator would then invoke the `SystemCall` method with the parameter class downcast to the parent `SysCallParmaters` class.

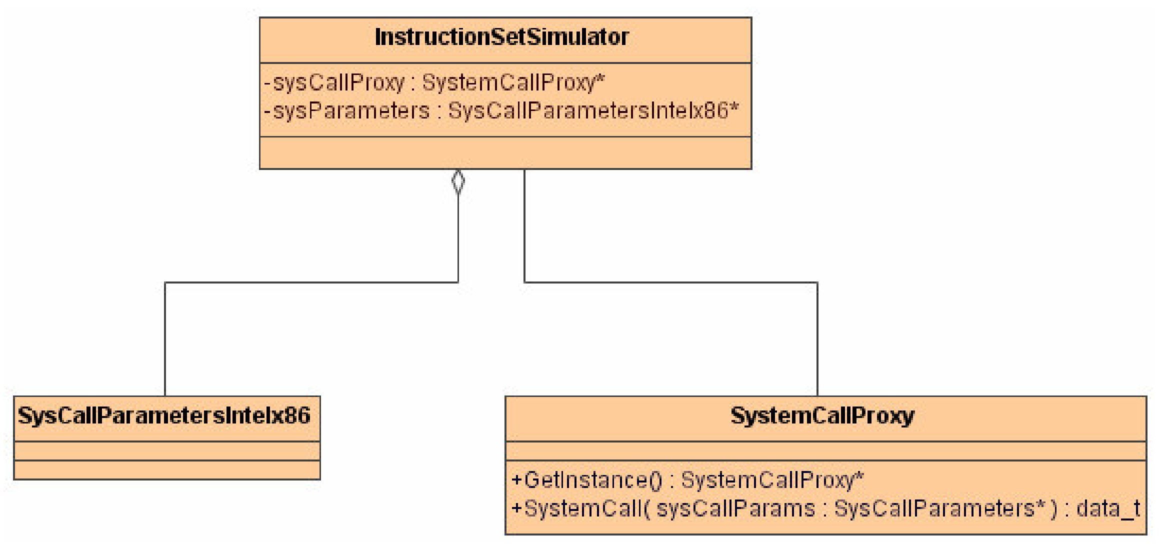
InstructionSetSimulator
-sysCallProxy : SystemCallProxy*
-sysParameters : SysCallParametersIntelx86*
SysCallParametersIntelx86
SystemCallProxy
+GetInstance() : SystemCallProxy*
+SystemCall(sysCallParams : SysCallParameters*) : data_t

Figure 2. Instruction Set Simulator class diagram

The sequence used to emulate system calls within the RITSim framework works in the following way:

1. The instruction set simulator traps the system call and converts the target registers to the generic system call registers through the method provided in the register class.
2. The instruction set simulator calls `SystemCall (*sysParam)` to invoke the emulated system call.
3. The system call proxy extracts the system call number from the `sysParam` object to determine the system call that will be executed.
4. The system call proxy extracts the needed register values for the system call and copies any needed memory to a local buffer.
5. The system call proxy invokes the system call on the host machine.

6. On completion of the host system call the system call proxy copies any memory buffer data to the target memory space and copies any return values into the correct generic system call registers.
7. Control returns to the instruction set simulator, which then extracts the resulting values from the register class back to the target registers.

The following sequence diagram illustrates the sequence used to execute the read system call.

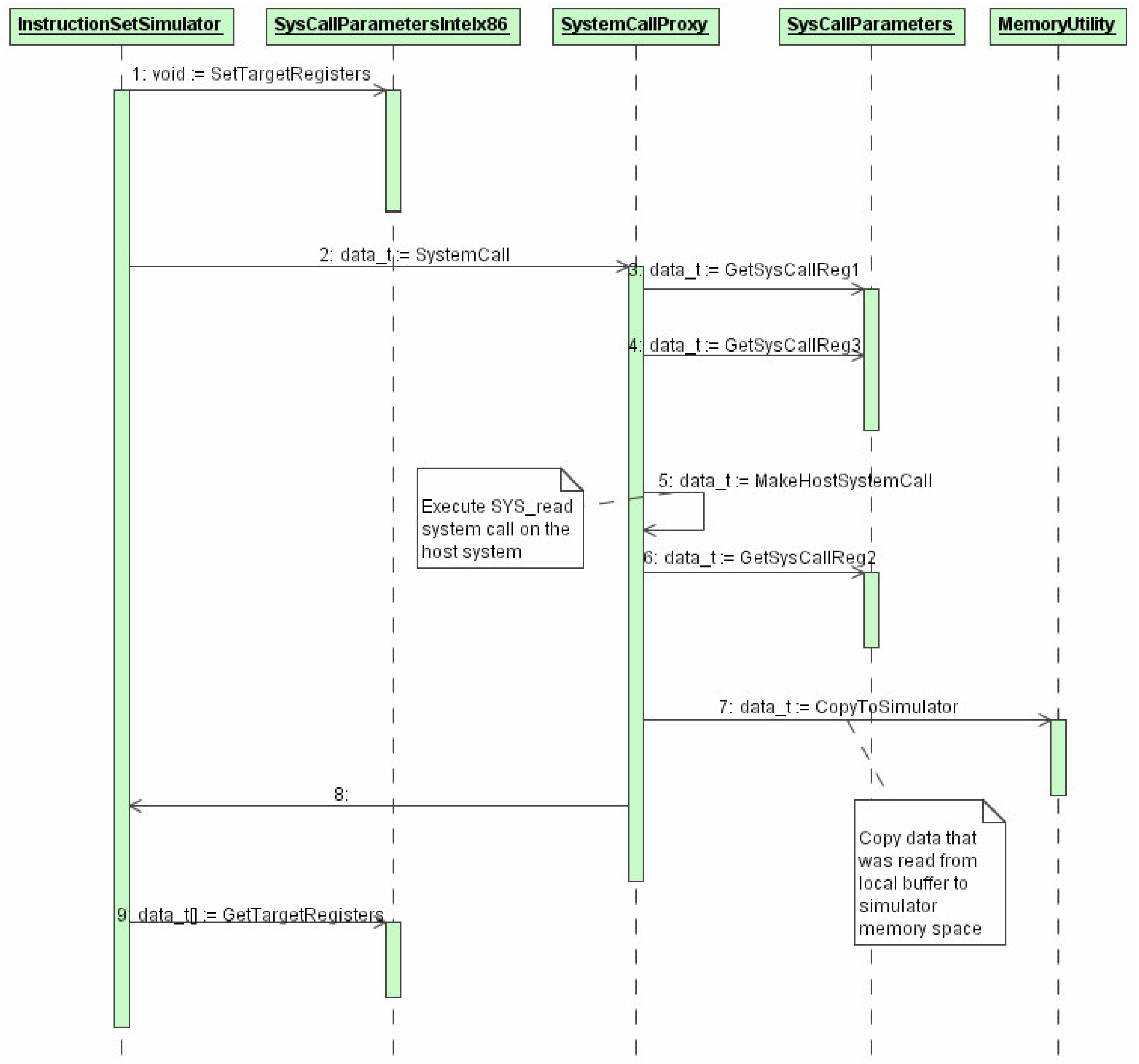
InstructionSetSimulator
SysCallParametersIntelx86
SystemCallProxy
SysCallParameters
MemoryUtility
1: void := SetTargetRegisters
2: data_t := SystemCall
3: data_t := GetSysCallReg1
4: data_t := GetSysCallReg3
5: data_t := MakeHostSystemCall
Execute SYS_read system call on the host system
6: data_t := GetSysCallReg2
7: data_t := CopyToSimulator
8:
Copy data that was read from local buffer to simulator memory space
9: data_t[] := GetTargetRegisters

Figure 3. Sequence diagram for the read system call

4.2. The Binary Application Loader Design

The design of the Binary Application Loader for RITSim will pull from the Linux procedure, the SimpleScaler [5] approach and the ELF specification. As discussed in section 3 the only executable format that will be supported is the Executable and Linking Format (ELF). ELF was selected because it is the standard format used by Linux. In addition to this it is also widely used in the Unix world.

Another design decision that was made was to only support static linking. There are many reasons for this; the primary reason is to maintain as much repeatability as possible. If dynamic linked libraries are used there is no guarantee that a test run in the future will reproduce the same result as the original simulation. This is simply because different versions of the shared libraries may be used over a period of time, even though the executable remains the same. Furthermore, anytime the executable is compiled on another machine running a different Linux image there is the chance that the performance will change due to differences in shared libraries. However, when using static linked executables all of the needed libraries are included in the executable. This greatly enhances the repeatability over time by removing the dependency on shared libraries. The statically linked executable will also be more stable when executed on different machines. Another significant benefit is the simplification of the binary loader code. With statically linked libraries there is no need to load an interpreter to handle the shared libraries. In Linux the shared library may be a format other than ELF. Some shared libraries are still in the a.out format. This further complicates things by needing not only an ELF interpreter, but an a.out interpreter. In addition to the interpreter issue there are also the added steps of finding, then mapping the shared libraries so they may be used by the executable.

The downside to using statically linked executables is the size. When the executable is compiled with statically linked libraries all of the libraries are included in the final executable, increasing the file size substantially. An example is this simple C program:

```
>cat hello.c
Main()
{
   printf("Hello world!!!\n");
}
```

When compiled as a dynamically linked executable that uses shared libraries the final executable size is 11542 Bytes. The statically linked executable was 423442 Bytes, which is nearly 37 times larger.

To compile a file in the correct format use a gcc compiler of version GCC-2.7.X or newer with the `-static` flag set. These versions of gcc default to creating output files in the ELF format. The `-static` flag is used to force the compiler to create a statically linked executable. To compile the simple `hello.c` file referenced above use the following command:

```
>gcc -static -o hello hello.c
```

The following command may then be used to verify that the executable is in the ELF format and is statically linked:

```
>file hello
```

The following is an example of what is displayed following the file command:

```
>ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for
GNU/Linux 2.2.5, statically linked, not stripped
```

This information is useful for insuring the executable that will be loaded by the binary application loader meets the loader's specifications, specifically that it is in the correct format, it is an ELF file, and it is statically linked. This is the only restriction placed on binaries that can be used by the loader and hence RITSim, i.e., any compiler (for any language) that can produce a statically linked ELF binary image can be simulated.

The RITSim binary application loader will use the following sequence to load the executable and prepare the stack for execution:

1. Perform consistency checks on the magic numbers. Check for ELF format and statically linked libraries.
2. Evaluate the ELF header to find text (code) and data segments and their sizes
3. Copy the text segment to simulator memory space.
4. Copy the data segment to simulator memory space.
5. Zero the `bss` segment.
6. Setup a new user space stack
7. Set the `start_code`, `end_code`, `start_data`, `end_data`, `start_brk`, `brk`, `start_stack`, `env_start`, `env_end`, `arg_start` and `arg_end` fields of the simulators memory descriptor.
8. Configure the target instruction pointer, the stack pointer, and the program counter registers.
9. On success return 0.

Notice that once again a mechanism will be needed to modify some of the target processor registers. To achieve this the instruction set simulator will use accessor methods provided by the `BinaryLoaderParameters` class to retrieve the modified values of the stack pointer, instruction pointer, and program counter registers. This approach allows the binary application loader to support different architectures without modification, which is similar to the approach used by the system call emulator. This allows the instruction set simulator to be modified to support different architectures without requiring a rework of the system call emulator or binary application loader code.

The following method is provided by the BinaryLoader class to load the ELF executable and prepare the user mode stack for execution within the simulator memory space:

```
data_t  LoadELF(BinaryLoaderParameters*)
```

The `BinaryLoaderParameters` class stores all of the data needed by the BinaryLoader class to load the ELF executable and prepare the stack. There is a complete description of this class at the end of this section.

The `LoadELF()` method performs the nine steps listed above to load the ELF executable and prepare it for execution. This process begins by checking the magic numbers to verify that the executable is in the correct format for the loader. The first step is to read the magic numbers that identify the executable as an ELF file. The very next step must be to verify that the executable is in a 32-bit format, as opposed to 64-bit. Checking the magic numbers up to the format entry will work for either 32-bit or 64-bit formats, however, subsequent magic numbers will be incorrect if the executable is in a 64-bit format and the header was stored incorrectly in the 32-bit header structure. The final magic number that is checked is to verify that it is an executable.

There are no magic numbers that identify the file as a dynamically or statically linked executable. The method used to verify that the executable is statically linked is to search through all of the program headers looking for an interpreter program type. This is a very general, robust method, because if there is an interpreter the executable must have a reference to a dynamically linked library. Conversely if there are no program headers that correspond to an interpreter the executable is statically linked.

The program headers that correspond to the text and data segments are located by searching for `p_flag` fields that correspond to either `read+execute` for the text segment or `read+write` for the data segment. Once these are identified the start addresses and sizes of the text and data segments can be extracted. Another field of importance in the text segment and data segment program headers is the `p_align` field. This specifies the value to which the segments are aligned in memory and in the file. When copying in the text and data segments these values must be used to ensure that the start address and size of the segment fall on these boundaries. If they do not, the end of the text segment must be padded with the beginning of the data segment, and the beginning of the data segment is padded with text data [28]. Since this was discussed in the ELF format specification, support was added to the binary application loader; however experimentation with a GCC version 3.2.2 compiler on a Linux 2.4.20-8 Kernel revealed that the data and text segments always fell on the `p_align` boundaries. Even though that may be the typical case within Linux, the loader supports this feature for non-aligned segments.

In some instances the text and data segments are combined into a single segment. This is identified by a program header that has the `p_flag` field set to `read+write+execute`. This case is handled by loading the entire segment. This segment begins with the text segment followed by the data segment. The program header `p_offset` field defines the start of the text segment. The start of the uninitialized data section (bss) is defined by the `p_memsz` field of the program header. The size of the bss is determined by subtracting the `p_memsz` field from the `p_filesz` field. These values are extracted the same whether the ELF file contains separate text and data segments or a single combined text and data segment. The start of the data section is determined differently depending on the executable. For the case with a separate text and data segment the start of the data segment is extracted from the `p_offset` field of the program header corresponding to the data segment, which is the same approach as finding the start of the text segment. When the text and data segments are lumped together this approach is no longer valid. The method used in this case is to search through each of the section headers looking for the section headers named either `.data` or `.data.init`. If both of these section names are found, the one with the lowest offset is used to set the start of the data segment and the end of the text segment. Decoding the section names takes a little extra work. The section names are defined in a string table, which contains a series of `NULL` terminated strings. The main ELF header `e_shstrndx` field defines the base of this table. Adding the section header `sh_name` field to the main ELF header `e_shstrndx` field defines the offset to the section name within the string table.

The `MemoryUtility` class is used to copy the text and data segments into simulator memory space. The method that copies the segment data uses `malloc()` to allocate memory, and then uses `fread()` to copy the segments from the file to the allocated space. Upon successful completion the starting address of the allocated buffer is returned, a zero is returned on failure. The starting address of the buffer is needed later to set various memory descriptor values. After the data segment has been copied the un-initialized data (bss) must be set to zero. This start of the bss section is located at the end of the data segment and is defined by the p_filesz field in the program header.

The stack is allocated through a call to the MemoryUtility class. The starting stack size is one page (4 Kbytes). This can be adjusted if needed by increasing the number of pages value in the BinaryLoader constructor. The stack is then configured by copying the environment variable strings (envp), command-line arguments (argv), the pointer to the environment variables strings, the pointer to the command-line arguments and finally the number of command-line arguments (argc) to the stack buffer. In Linux the stack grows from the bottom to the top, so the first data copied to the stack ends at the end of the new stack and grows from there. Figure 4 illustrates the bottom of the user mode stack.

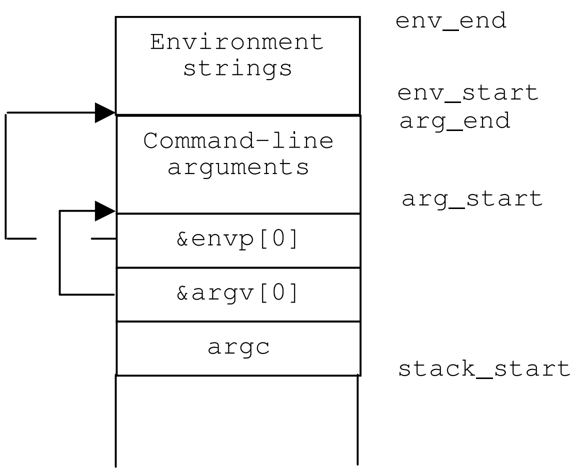

Figure 4. Bottom of the user mode stack

After the stack is set, some of the memory descriptors must be updated prior to execution of the program. These values are located in the mm_struct structure defined in sched.h. The start_code and end_code entries are updated with the start and end addresses of the text segment. These are determined by the start of the text segment buffer returned by the MemoryUtility class and the size of the text segment extracted from the text segment program header. The start_data and end_data entries are updated with the start and end address of the data segment. The values for these are obtained in the same manner as the text segment values. The `start_brk` and `brk` entries are updated with the start of the heap and the current location of the heap pointer. These values are communicated to the binary loader from the instruction set simulator through accessor methods in the `BinaryLoaderParameters` class. The `start_stack`, `env_start`, `env_end`, `arg_start` and `arg_end` all relate to the user mode stack and are shown in figure 4. These values are calculated based on the starting address of the stack returned from the `MemoryUtility` class upon allocation, the size of the stack, and the size of the `envp` and `argv` arrays. During testing it was discovered that the `mm_struct` is only visible from kernel level code. Since this code is resident in user space a different solution was needed. To handle this a new structure was created named `MemoryStruct`. This structure contains all of the above-mentioned fields, and was added to the `BinaryLoaderParameters`. The instruction set simulator accesses this structure through the `GetMemoryDescriptor` method in the `BinaryLoaderParameters` class.

The final step is to set the instruction pointer register to point to the start of the text segment, to set the stack pointer register to point to the start of the user mode stack, and to set the program counter to the virtual address that defines the entry point that the system first transfers control to start the new process. The binary application loader sets these registers in the `BinaryLoaderParameters` object. Once the executable is loaded the instruction set simulator must get the updated register values through the supplied accessor methods in the `BinaryLoaderParameters` class.

Another method provided by the `BinaryLoader` class that is useful for debug is:

```
void CoreDump(char* addr, data_t size);
```

It is very similar to the standard core dump. The major difference is that it dumps the loaded executable to `stdout` instead of to a file named core. If desired the output may be redirected to a file and viewed to check the data and memory locations of the copied information.

As stated previously in this section the `BinaryLoaderParameters` class contains all of the data needed by the `LoadELF()` method. The executable file name is accessible through the following methods:

```
void  SetFileName(char*)
char* GetFileName()
```

The calling method must set the file name, including the path, prior to calling. The instruction pointer, stack pointer, and program counter registers are accessible through the following methods:

```
void   SetInstructPointerReg(data_t)
void   SetStackPointerReg(data_t)
void   SetProgramCounterReg(data_t)
data_t GetInstructPointerReg()
data_t GetStackPointerReg()
data_t GetInstructPointerReg()
```

The `LoadELF()` method will set the instruction pointer, stack pointer, and program counter registers at the completion of the method call. The instruction set simulator will then need to use the get methods to retrieve the updated copy of these registers. The environment variables and command line arguments are accessible through the following methods:

```
void SetArgC(int)
void SetArgV(char**)
void SetEnvP(char**)
int    GetArgC()
```

```
char** GetArgV()
char** GetEnvP()
```

The calling method is responsible for setting the `ArgC` and `ArgV` values, while the `EnvP` value is optional. The heap information is accessible through the following methods:

```
void SetHeapStart(data_t)
void SetHeapCurrent(data_t)
data_t GetHeapStart()
data_t GetHeapCurrent()
```

Once again the calling method is responsible for setting the heap start address and the current heap address. These are needed by the `LoadELF()` method to set the heap fields in the memory descriptor structure.

The following class diagram illustrates the methods associated with each class required by the binary loader and their relationships see figure 5.

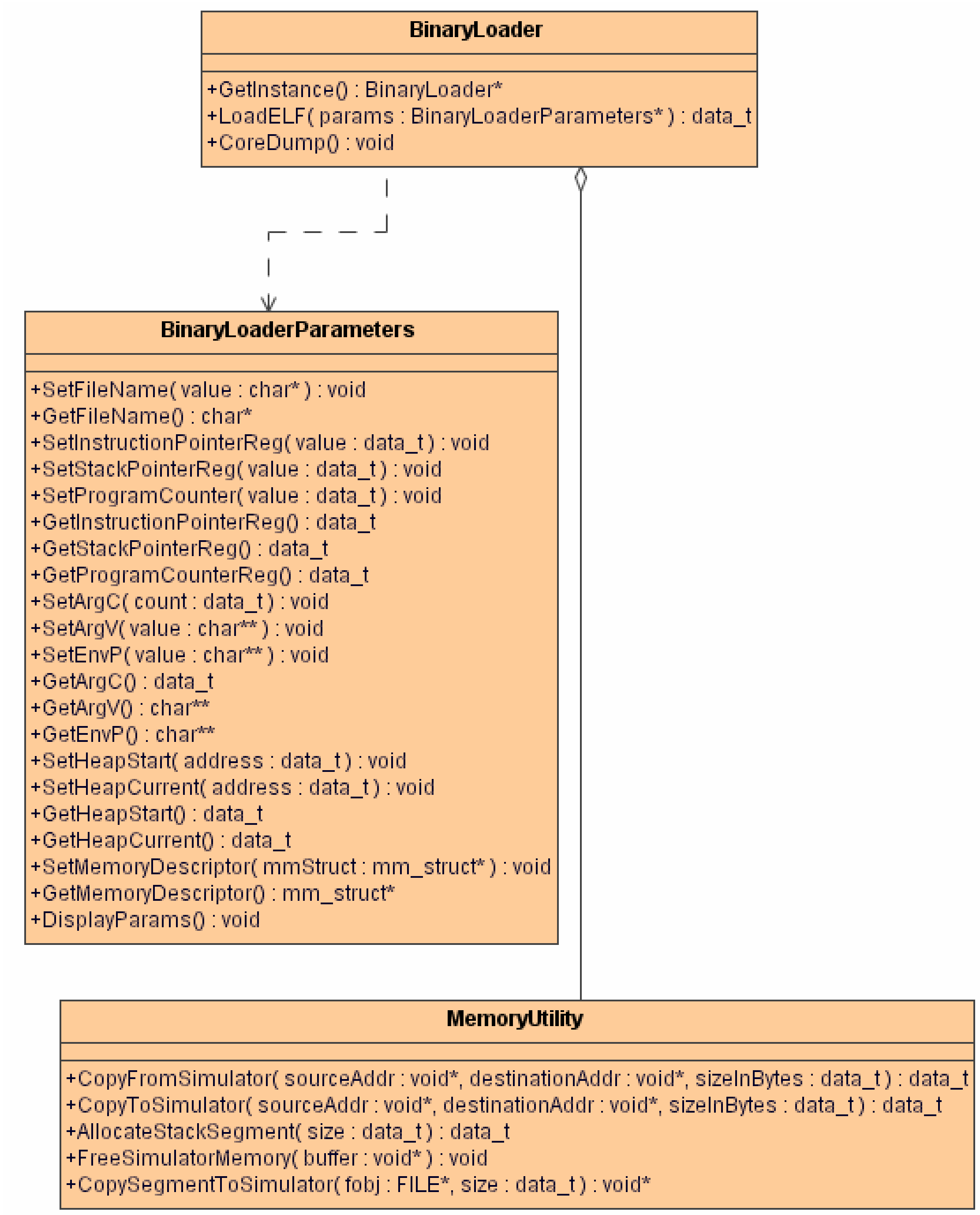
BinaryLoader
+GetInstance() : BinaryLoader*
+LoadELF(params : BinaryLoaderParameters*) : data_t
+CoreDump() : void
BinaryLoaderParameters
+SetFileName(value : char*) : void
+GetFileName() : char*
+SetInstructionPointerReg(value : data_t) : void
+SetStackPointerReg(value : data_t) : void
+SetProgramCounter(value : data_t) : void
+GetInstructionPointerReg() : data_t
+GetStackPointerReg() : data_t
+GetProgramCounterReg() : data_t
+SetArgC(count : data_t) : void
+SetArgV(value : char**) : void
+SetEnvP(value : char**) : void
+GetArgC() : data_t
+GetArgV() : char**
+GetEnvP() : char**
+SetHeapStart(address : data_t) : void
+SetHeapCurrent(address : data_t) : void
+GetHeapStart() : data_t
+GetHeapCurrent() : data_t
+SetMemoryDescriptor(mmStruct : mm_struct*) : void
+GetMemoryDescriptor() : mm_struct*
+DisplayParams() : void
MemoryUtility
+CopyFromSimulator(sourceAddr : void*, destinationAddr : void*, sizeInBytes : data_t) : data_t
+CopyToSimulator(sourceAddr : void*, destinationAddr : void*, sizeInBytes : data_t) : data_t
+AllocateStackSegment(size : data_t) : data_t
+FreeSimulatorMemory(buffer : void*) : void
+CopySegmentToSimulator(fobj : FILE*, size : data_t) : void*

Figure 5. BinaryLoader Class Diagram

The following sequence diagram illustrates the calls that are made between the calling method in the instruction set simulator, the `BinaryLoader` class and the supporting classes required to complete the `LoadELF()` method, see figure 6.

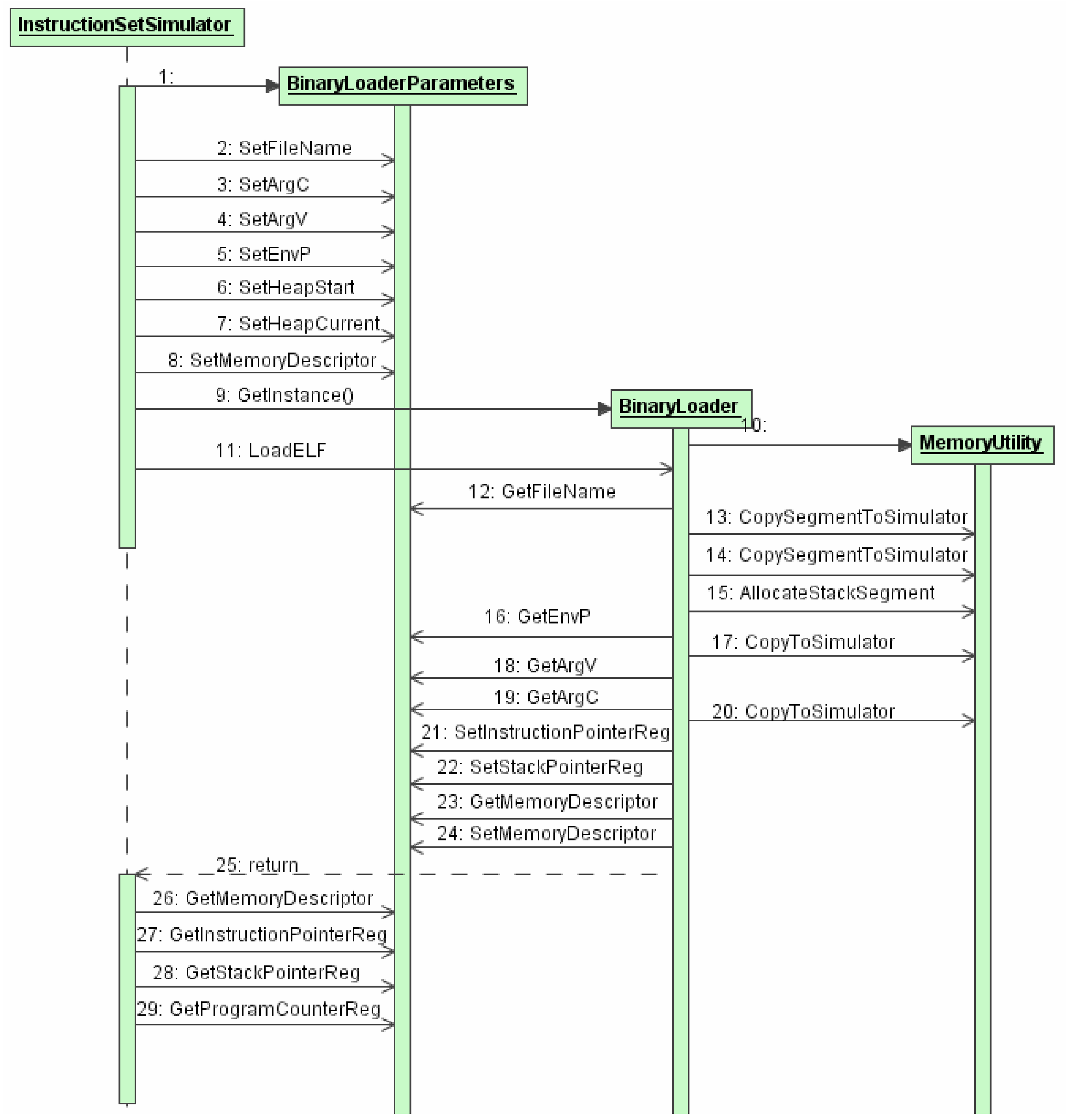

Figure 6. BinaryLoader Sequence Diagram

4.3. The Linux Virtual Environment Design

There were several reasons why User Mode Linux was selected for this work. The primary reason was that it provides a complete Linux kernel that runs in a virtual environment. Furthermore it is a patch that is applied to a kernel not a stand-alone piece of software. This provides an important and often-overlooked feature, which is a seamless upgrade path. Take the case of moving from 2.4 to 2.6 kernels; by applying the correct patch to the newer kernel you now have the latest Linux kernel to run simulations on. In fact, User Mode Linux is now included in the most recent version of the Linux Kernel. Any 2.6.9 or newer kernel includes the User Mode Linux source code so no patch will be needed. User Mode Linux also allows the user to run a different kernel version than the physical machine is running. The significance of this is that simulations maybe run against the latest Linux kernels without reconfiguring the entire host system to also support the latest kernel. Conversely, perhaps the user would like to compare the simulation results of new architecture against previous results obtained some time ago. This new simulation may be run on any machine running any kernel, quite possibly with a newer kernel installed, with the same version of the virtual operating system that was used to generate the previous results. Lastly it's open source code available under the general public license (GPL). This approach provides researchers with the freedom to simulate architectures on the latest kernels or return to older kernels for comparisons to previous simulations.

Within RITSim, the virtual operating system portion must provide an environment that loads a Linux kernel and executes an application or set of applications within a virtual environment. As supplied, User Mode Linux provides the Linux kernel running within a virtual environment. This leaves the task of executing the applications within that environment. Linux makes use of startup scripts during the initialization process to prepare the system for use. Linux also supports various run levels, or system states, that are used to control the state of the system. Typical Linux run levels are:

```
0 - Halt
1 - Single user mode
2 - Multi-user mode, without network support.
3 - Full Multi-user mode
5 - X11 mode
6 - reboot
```

The run level determines the startup scripts that will be called to initialize the system, or in the case of the halt and reboot run levels, the kill scripts that will be called to halt the system. This initialization sequence begins with the creation of the `init` process. The `init` process is spawned at the start of the system and is the parent process to all subsequent processes. When booting to run levels 2, 3 or 5 the first script run by the `init` process is the `/etc/rc.d/rc.sysinit` script. This script is responsible for many functions including starting the virtual memory swapping, checking and mounting the root files system, checking and mounting other file systems, setting the system clock, initializing serial ports, etc. The init process then executes the scripts particular to the selected run level by parsing through the correct `rcX.d` directory, where `X` is the run level. In the case of run levels 2, 3, or 5 the last script that is executed is typically the `rc.local` script. This script is where custom initialization processes may be defined. It is at this point that the simulation applications may be run. If desired an application can be started at bootup by adding the call to run the application in the this script.

Since User Mode Linux boots using the same mechanisms as the normal Linux kernel, modifications to the `rc.local` script can be utilized to execute the desired applications once the virtual kernel has finished the boot process. Furthermore, `init 0` maybe called at the end of the `rc.local` script to halt the system after the simulation has completed.

Chapter 5 Results

5.1. *The Linux OS Emulator*

There were two main portions of the emulator. The register class that handles the conversion of the target register set to a generic register set used to invoke the system call and the class that parses the system call number and makes the system call on the host machine.

The register class was designed with a parent class that would give a layer of abstraction for the emulator software from the target register set. The parent class contains the methods to extract the generic register values prior to system call execution as well as the methods to update the generic registers after completion of the system call. The methods used to either copy the target register values to the generic register set or to copy back the generic register values to the target register values are abstract methods realized in the child class. Each of the target architectures will have a new child class that handles the mapping of the target registers. The greatest benefit to this approach is that the emulator only deals with the parent class; therefore the emulator code does not need to be modified to handle the various child classes associated with different architectures. The instruction set simulator uses the methods implemented in the child class, which is fine since the instruction set simulator is specific to the target architecture. This approach worked very well for this project.

The class that is responsible for decoding the system call and making the system call on the host system was created as designed. An effort was made to emulate all system calls, however, this proved to be an unrealistic proposal. While most were successfully emulated, there were a few that were not appropriate to emulate at this level, some that were too new to emulate on the system used for development, and some that were no longer supported in the 2.4 kernel.

The list of obsolete system calls that were not implemented in the 2.4 kernel and therefore not emulated are: `afs_syscall`, `break`, `ftime`, `getpmsg`, `gtty`, `lock`, `mpx`, `prof`, `profil`, `putpmsg`, `security`, `stty`, `ulimit`, and `oldolduname`. The `nfsservctl` system call was emulated, but it is currently commented out, the include file needed for the definition of the setup structure clashes with other include files.

The system calls that were deemed not appropriate to emulate were `reboot`, `clone`, `ipc`, `minicore`, `madvise`, `and futex`. `Reboot` was not implemented since the simulation software should never call reboot. `Clone` was not emulated since it deals with pointers to functions in the child process and with the child stack, this would need to be handled by the instruction set simulator. `ipc` is a common kernel entry point for the System V IPC calls, it was not implemented since user programs should not use this call but should use the appropriate individual calls instead. `minicore` requests a vector describing which pages of a file are in core and can be read without disk access, it was not implemented since at this stage the simulator has no knowledge of this on the host system. The `madvise` system call advises the kernel about how to handle paging input/output in the address range. For this call to be successful the simulator would have to know what memory areas in the host to advise the kernel about, therefore it was not implemented since this would not be the case. The `futex` system call provides a method for a program to wait for a value at a given address to change, and a method to wake up anyone waiting on a particular address. Again this would require knowledge of the host memory map.

The number of system calls supported by newer Linux kernels is growing at an alarming rate. The 2.2 version of the kernel supported 190 system calls. Version 2.4.2 added system calls numbered from 191 to 219, version 2.4.18 renamed the system calls numbereed 191 to 219 and added new system calls with numbers from 220 to 237. Version 2.4.19 added new system calls from 238 to 242. Version 2.4.20 added new system calls from 243 to 252. The 2.6 release featured the addition of new system calls from 253 to 271, while 2.6.2 added two more from 272 to 273. The problem with this is that older kernels do not have the system call definitions to compile the newer system calls. Due to this a line was drawn as to what system calls would be supported, the emulator currently supports up to system call 244. This was selected since the kernel used on the development machine, version 2.4.20-8, supported up to this call. Calls beyond this result in compilation errors due to undefined strutures and system call enumerations. This should be a good break point since most current Linux distributions use 2.4.2x kernels and should work with this emulator, note however that kernels older than 2.4.20 may not compile. If that is the case the offending system calls may simply be commented out and recompiled with only the supported system calls for that version. If the host machine is running a newer version of the kernel and the simulated software is attempting to use an unsupported system call the emulator will display a message that states that an unsupported system call was made. After the system call parameters have been determined support may simply be added to the class for the new system call on an as needed basis.

5.2. *The Linux Binary Application Loader*

Successfully opening and verifying the format of an executable that meets our requirements was the first step in the verification of the `LoadELF()` code. Recall that the executable must be an ELF file that is statically linked. After successfully completing this further testing included changing the magic numbers to unacceptable values to verify that incorrect file formats would be handled correctly. The final test was to attempt to load a dynamically linked ELF executable to test the method used to check for statically linked executables. This was successfully identified as well, with the loader printing a message and aborting the load.

The next test was to use the `CoreDump()` method to verify that the text segment was copied correctly into simulator memory space. This was verified by checking the core dump data with the data in the original file. The data segment was verified next in the same manner, with the additional step of checking that the un-initialized data section (bss) was correctly set to all zeros. Finally the stack segment was verified, all the `envp` and `argv` strings were copied correctly and the pointers were set correctly. The memory descriptors were then verified against the `CoreDump()` data to insure they were pointing to the right locations in the corresponding segments.

Another verification technique that was attempted was to compare the `CoreDump()` data with an actual core dump of the running program within Linux. To get a core dump of the test file an un-initialized pointer was referenced this caused a core dump when the program was executed. Comparing the core dump data showed that the initial header information was the same, data following that was different however. The most likely explanation for this is that the `LoadELF()` method copies the entire text, data and stack segments. The Linux binary loader on the other hand only copies the first page of each segment and sets up a page table to locate the subsequent pages when a page fault is encountered.

The current methods used by the `MemoryUtility` class to copy the segments into simulator memory space simply use `malloc()` to allocate the space needed to store the data. As RITSim matures the memory subsystem extended to provide a true model of simulated memory. This may be extended further to make use of a sparse file to map the simulated system memory. This feature would allow the simulator to map large amounts of simulated memory without actually allocating it all from the host system memory. Sparse files treat large sections of zeros as holes, thereby saving large amounts of space if most of the data is represented by large sections of zeros. This is in fact often the case with memory maps. This approach allows a researcher to simulate systems that contain large amounts of memory without requiring the same of the host machine.

One element that still needs to be handled once the RITSim infrastructure is further developed is the mapping of the virtual addresses to the text and data segments. In general executable programs rely on absolute addresses. To support this the virtual addresses of the text and data segments must match those in the executable file. This is handled in Linux by using the `mmap()` method to map the segments into virtual memory. At this stage of development the elements within RITSim that will handle this have not been completed. Once support for simulated virtual addressing has been enabled within RITSim this mapping will need to be addressed. The `p_vaddr` member of the program header structure contains the virtual addresses required by the segments. To execute correctly the virtual addresses of the loaded segments must match those defined in the `p_vaddr` member.

5.3. *The Virtual Linux Environment*

By executing a single command that loaded a virtual Linux environment, executed a simple application, and exited the virtual Linux environment successfully demonstrated the feasibility of this design. This environment was created by downloading the source code for a Linux kernel and the matching User Mode Linux kernel patch. User Mode Linux also needs a root file system to boot from, this also needs to be downloaded. All of this is available for download on *sourceforge.net*. The following packages were downloaded for this work:

Linux Kernel:

```
linux-2.4.27.tar.bz2
```

User Mode Linux Patch:

```
uml-patch-2.4.27-1.bz2
```

Root File System:

```
root_fs.md-7.2-server.pristine.20021012.bz2
```

Creation of the User Mode Linux executable begins by unpacking the Linux kernel source code. This is followed by applying the User Mode Linux patch using the following command:

```
>bzcat uml-patch-2.4.0-prerelease.bz2 | patch -p1
```

The next step was to run xconfig as follows:

```
>make xconfig ARCH=um
```

When prompted to select the configurations, just use the default values. The ARCH=um switch directs the configuration tool to build for User Mode Linux instead of an x86 or other physical architecture. After this is completed the executable is built like a normal kernel using the following command:

```
>make linux ARCH=um
```

The output of this is an executable called linux. The only remaining step is creating the root file system needed for User Mode Linux to boot. This is created by unzipping the downloaded root file system, in this case it is a Red Hat root file system. Once it is unzipped the name should be changed to root_fs, as this is the default name of the root file system that User Mode Linux tries to mount during boot up. The linux executable and the root_fs should be in the same directory.

At this point running the linux executable will boot a 2.4.27 kernel in a virtual environment. Modification of the rc.local script can be handled in one of two ways. The first is to boot the virtual Linux kernel and modify the rc.local script from there. The second option is to mount the root_fs file system and modify the rc.local script through the mount point.

The final step is to move the desired applications into the root_fs so they can be executed in the virtual environment. This is handled by mounting the root_fs from the host machine. Using the following command:

```
>mount root_fs mnt -o loop
```

Once mounted, the executables may be copied into the virtual systems root file system. A /sim directory was created off the root to place any executables that may be executed.

To test the design, a simple "hello world" program was copied into the /sim directory in the root_fs. The rc.local script in the root_fs was then modified to run the “hello world” executable, followed by the `init 0` command to halt the system. Once these were completed the `linux` command was run, the following screen capture illustrates the successful verification of the design:

```
Checking for the skas3 patch in the host...not found
Checking for /proc/mm...not found
Checking PROT_EXEC mmap in /tmp...OK
Checking for /dev/anon on the host...Not available (open failed with errno 2)
Checking for /dev/anon on the host...Not available (open failed with errno 2)
Checking for /dev/anon on the host...Not available (open failed with errno 2)
Checking for /dev/anon on the host...Not available (open failed with errno 2)
Linux version 2.4.27-1um (swarne01@chipotle9.eng.mc.xerox.com) (gcc version 3.2.2
20030222 (Red Hat Linux 3.2.2-5)) #28 Tue Mar 15 20:17:56 EST 2005
On node 0 totalpages: 8192
zone(0): 8192 pages.
zone(1): 0 pages.
zone(2): 0 pages.
Kernel command line: init 2 root=/dev/ubd0
Calibrating delay loop... 1589.24 BogoMIPS
Memory: 29052k available
Dentry cache hash table entries: 4096 (order: 3, 32768 bytes)
Inode cache hash table entries: 2048 (order: 2, 16384 bytes)
Mount cache hash table entries: 512 (order: 0, 4096 bytes)
Buffer cache hash table entries: 1024 (order: 0, 4096 bytes)
Page-cache hash table entries: 8192 (order: 3, 32768 bytes)
Checking for host processor cmov support...Yes
Checking for host processor xmm support...No
Checking that ptrace can change system call numbers...OK
Checking syscall emulation patch for ptrace...missing
Checking that host ptys support output SIGIO...Yes
Checking that host ptys support SIGIO on close...No, enabling workaround
POSIX conformance testing by UNIFIX
Linux NET4.0 for Linux 2.4
Based upon Swansea University Computer Society NET3.039
Initializing RT netlink socket
Starting kswapd
VFS: Disk quotas vdquot_6.5.1
Journalled Block Device driver loaded
devfs: v1.12c (20020818) Richard Gooch (rgooch@atnf.csiro.au)
devfs: boot_options: 0x1
JFFS version 1.0, (C) 1999, 2000  Axis Communications AB
JFFS2 version 2.1. (C) 2001 Red Hat, Inc., designed by Axis Communications AB.
pty: 256 Unix98 ptys configured
SLIP: version 0.8.4-NET3.019-NEWTTY (dynamic channels, max=256).
RAMDISK driver initialized: 16 RAM disks of 4096K size 1024 blocksize
loop: loaded (max 8 devices)
PPP generic driver version 2.4.2
Universal TUN/TAP device driver 1.5 (C)1999-2002 Maxim Krasnyansky
SCSI subsystem driver Revision: 1.00
scsi0 : scsi_debug, Version: 0.61 (20020815), num_devs=1, dev_size_mb=8, opts=0x0
  Vendor: Linux     Model: scsi_debug        Rev: 0004
  Type:   Direct-Access                      ANSI SCSI revision: 03
blkmtd: error: missing `device' name

Initializing software serial port version 1
mconsole (version 2) initialized on /home/swarne01/.uml/wMjdvv/mconsole
Partition check:
 ubda: unknown partition table
UML Audio Relay (host dsp = /dev/sound/dsp, host mixer = /dev/sound/mixer)
Initializing stdio console driver
NET4: Linux TCP/IP 1.0 for NET4.0
IP Protocols: ICMP, UDP, TCP
IP: routing cache hash table of 512 buckets, 4Kbytes
TCP: Hash tables configured (established 2048 bind 4096)
NET4: Unix domain sockets 1.0/SMP for Linux NET4.0.
VFS: Mounted root (ext2 filesystem) readonly.
Mounted devfs on /dev
INIT: version 2.78 booting

                      Welcome to Red Hat Linux
               Press 'I' to enter interactive startup.
Mounting proc filesystem:  [  OK  ]
Configuring kernel parameters:  [  OK  ]
Setting clock : Fri Mar 18 17:25:10 EST 2005 [  OK  ]
```

```
Activating swap partitions:  [  OK  ]
Setting hostname redhat72.goober.org:  [  OK  ]
Checking root filesystem
/dev/ubd/0: clean, 38766/64768 files, 213332/259072 blocks
[/sbin/fsck.ext2 (1) -- /] fsck.ext2 -a /dev/ubd/0
[  OK  ]
Remounting root filesystem in read-write mode:  [  OK  ]
Finding module dependencies:  depmod: cannot read ELF header from
/lib/modules/2.4.27-1um/modules.dep
depmod: cannot read ELF header from /lib/modules/2.4.27-1um/modules.generic_string
depmod: /lib/modules/2.4.27-1um/modules.ieee1394map is not an ELF file
depmod: /lib/modules/2.4.27-1um/modules.isapnpmap is not an ELF file
depmod: cannot read ELF header from /lib/modules/2.4.27-1um/modules.parportmap
depmod: /lib/modules/2.4.27-1um/modules.pcimap is not an ELF file
depmod: cannot read ELF header from /lib/modules/2.4.27-1um/modules.pnpbiosmap
depmod: /lib/modules/2.4.27-1um/modules.usbmap is not an ELF file
[FAILED]
Checking filesystems
Checking all file systems.
[  OK  ]
Mounting local filesystems:  [  OK  ]
Enabling local filesystem quotas:  [  OK  ]
swapon: cannot stat /dev/ubd/1: No such file or directory
Enabling swap space:  [  OK  ]
INIT: Entering runlevel: 2
Entering non-interactive startup
Setting network parameters:  [  OK  ]
Bringing up interface lo:  [  OK  ]
SIOCADDRT: No such device
SIOCADDRT: Network is unreachable
Starting system logger: [  OK  ]
Starting kernel logger: [  OK  ]
Initializing random number generator:  [  OK  ]
Starting sshd:  [  OK  ]
Starting sendmail: [  OK  ]
Starting crond: [  OK  ]
Running Linuxconf hooks:  [  OK  ]

hello world!!!

INIT: Switching to runlevel: 0
INIT: Sending processes the TERM signal
Stopping sshd:[  OK  ]
Shutting down sendmail: [  OK  ]
Stopping crond: [  OK  ]
Saving random seed:  [  OK  ]
Shutting down kernel logger: [  OK  ]
Shutting down system logger: [  OK  ]
Starting killall:  [  OK  ]
Sending all processes the TERM signal...
Sending all processes the KILL signal...
Syncing hardware clock to system time
Turning off quotas:
umount2: Device or resource busy
umount: devfs: not found
umount: /dev: Illegal seek
Halting system...
Power down.
tracing thread pid = 11077
```

Notice the boot sequence is the same as a normal Linux kernel. As the boot sequence progresses the message "hello world" is printed signaling that the `rc.local` script was called. This in turn executed the simple "hello world" application. Directly following that the system enters run level 0 which is the halt run level. The end of the screen capture shows the system going through the normal halt sequence as defined by the scripts in the `/etc/rc.d/rc0.d` directory.

Chapter 6 Discussion

6.1. *The Linux OS Emulator*

As designed, the Linux operating system emulator provides the ability to support different target architectures without requiring modifications to the system call proxy layer. This is a unique solution not seen in existing simulator implementations. Furthermore, an attempt was made to emulate all of the Linux system calls. In the end the majority of the system calls supported by the 2.4.20-8 kernel were emulated. The calls that were not supported include calls that require knowledge of memory or devices outside the scope of the emulator, obsolete calls, and calls that should not come from the simulator, such as `reboot()`. The existing simulators that were studied support only a subset of the system calls supported by the operating system. This limits them to supporting only certain test suites. By contrast the emulator within RITSim gives a researcher the freedom to execute nearly any program within the simulation environment by emulating all of the applicable system calls provided by the Linux kernel.

6.2. *The Linux Binary Application Loader*

The binary application loader is responsible for loading a statically linked, ELF executable into simulated virtual memory and for creating a new user mode stack to prepare the new process for execution. This was designed based on the ELF documentation and the existing Linux ELF loader. The major difference from the existing Linux solution was the need to load the executable into simulated virtual memory instead of simply mapping into the systems virtual memory. This design was demonstrated by loading a simple executable as well as the User Mode Linux executable. Both of these applications were successfully loaded, verification was completed by comparing a core dump of the loaded executable with the original executable to insure that all of the segments were loaded correctly. The memory descriptors that map the locations and sizes of the segments were also updated and verified.

There is still one piece that is missing that needs to be addressed as the RITSim environment matures, that is the mapping of text and data segments to the correct simulated virtual address. As compiled, statically linked ELF executables contain absolute code. The implication of this is that the text and data segment virtual addresses must match those used to build the executable. These are defined in a descriptor in the header of each section. At this point no extra support is needed in the binary application loader, but the memory utility class will need to be expanded to provide this support. Once the mechanism to support simulated virtual memory has been added to RITSim, this must be addressed in the binary loader for the code to execute correctly. It is worth noting that if an application is simulated within the virtual Linux environment then this isn't needed. That is, if the absolute addresses of the Linux application are correct, any virtual translation (and the associated OS code to support that) is handled entirely within UML.

6.3. The Virtual Linux Environment

As demonstrated in the results section, the selected method to provide a fully functional Linux operating system running in a virtual environment has many benefits. The first of which is that we were able to modify User Mode Linux to run applications within a virtual environment without any modifications to the kernel or to the User Mode Linux source code. Furthermore, depending on the situation there are multiple was to modify the `rc.local` script. The first is by booting User Mode Linux and modifying the file within the root file system. The second method involves mounting the virtual environments root files system from the host machine and modifying the `rc.local` script from the host machine. To modify the `rc.local` script from within the virtual environment simply invoke the User Mode Linux executable as follows:

```
> linux init 1
```

This boots the virtual Linux kernel to run level 1, which is the single user mode. When booting to this run level, the kernel does not run the `rc.local` script. This allows the user to modify the script even if past modifications instructed the kernel to run an application and shutdown. If the virtual kernel boots to run levels 2, 3, or 5 after the `rc.local` script was modified, the system will automatically halt after running the specified executable(s), so the `rc.local` script cannot be edited when booted to these run levels. This further allows simulated environments to be created within the root_fs scripts, which are completely repeatable.

Verification of this design was completed by running a single application. Normal simulations, however, often rely on running a suite of applications to test the architecture performance under different workloads. There are a few ways this could be supported with this approach. This first is to simply add the required executable calls to the `rc.local` script. Another, more structured approach would be to call another script or set of scripts from `rc.local`. This would be a useful approach to help categorize and manage large sets of executables. Yet another approach would be to make use of the make command to execute multiple applications.

Chapter 7 Summary and Future Work

7.1. Summary

This work has demonstrated three components of the RITSim simulation environment, a Linux operating system emulator, a binary application loader and a Linux operating system running in a virtual environment. The first component that was demonstrated was an easily adaptable Linux operating system emulation tool. This tool provides a first order simulation environment that emulates Linux system calls on a host machine. The method used to interface with the instruction set simulator allows the target architecture to change without requiring changes to the Linux emulator. This gives researchers the freedom to quickly simulate new architectures.

The binary application loader was created to load executables that will be simulated into the simulators virtual memory space. The scope of this work included loading statically linked, ELF executables into simulated virtual memory space and preparing the user mode stack for the new process.

User Mode Linux was utilized to provide a fully functional Linux operating system running in a virtual environment. Furthermore a method was demonstrated that allows a user to boot a virtual Linux operating system, execute an application and exit from a single command. Of significant note is that this method required no modifications to the Linux kernel or the User Mode Linux patch. In fact it can be configured to run different applications or groups of applications without requiring a rebuild of the executable.

The goal for RITSim is to provide a simulation environment that will combine fast, first order simulations with detailed, highly accurate full system simulations. The components demonstrated in this work provide key elements that are required to realize that goal.

7.2. Future Work

Areas that may be addressed in the future on the emulator include integrating the instruction set simulator and expanding the memory utility to support simulated virtual memory. Another possible area of improvement is expanding the memory utility to interact with the cache simulator to provide increased levels of timing accuracy. Additional work may also be required if the host machine moves to a 2.6 kernel, in this case support may need to be added for newer system calls. This should be straightforward since the emulator prints out a message if an unsupported system call is received that identifies the system call number. This is used to identify additional system calls that need to be supported.

In order to fully integrate the binary application into the RITSim environment support will have to be added for simulated virtual memory space. When an ELF file is loaded into the simulator the virtual address of the loaded executable must match the addresses of the segments in the ELF executable. At this point there is no support for simulated virtual memory in RITSim, once this is available the text and data segments will need to be mapped to the correct virtual addresses prior to executing the program.

The use of User Mode Linux to provide a full functional Linux kernel running in a virtual environment coupled with modifications to the `rc.local` script provides many desirable features for detailed, high accuracy simulation modes. Recall, however, that this concept was demonstrated by running a single executable, then exiting by halting the virtual kernel. To make this more useful, support needs to be added to run a series of applications such as one of the standard test suites. Suggestions were made on how this could be accomplished, such as the use of scripts or the make system.

These features will need to be addressed to help RITSim provide the seamless integration of fast, first order simulations with highly detailed, full system simulations within the same simulation environment.

Bibliography

[1] Understanding the Linux Kernel. Daniel P.Bovet and Marco Cesati. 2nd Edition, December 2002.

[2] SystemC Version 2.0 User's Guide

[3] Techniques for Implementing Fast Processor Simulators. M. Moudgill. In *The 31st Annual Simulation Symposium*, April 1998, Boston, Massachusetts.

[4] The Liberty Simulation Environment, Version 1.0. M. Vachharajani, N. Vachharajani, D. Penry, J. Blome, and D. August. In *Performance Evaluation Review: Special Issue on Tools for Architecture Research*, Volume 31, Number 4, March, 2004.

[5] The SimpleScaler Tool Set, Version 2.0. D. Burger and T. Austin. Technical Report 1342, University of Wisconsin-Madison Computer Sciences Department, June 1997.

[6] Full-System Timing-First Simulation. C. Mauer, M. Hill, and D. Wood. In *2002 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, June 15, 2002.

[7] User Mode Linux HOWTO. The User Mode Linux Core Team, http://user-mode-linux.sourceforge.net/UserModeLinux-HOWTO.html

[8] Programming with User Mode Linux. N. Weber, Linux Gazette, http://www.linuxgazette.com/issue90/weber.html

[9] Measuring Experimental Error in Microprocessor Simulation. R. Desikan, D. Burger, and S. Keckler. In Proceedings of the 28th annual international symposium on Computer architecture, pp. 266-277, 2001.

[10] Using Complete Machine Simulation to Understand Computer System Behavior. S. Herrod. Ph. D. Thesis, Department of Computer Science, Stanford University, February 1998.

[11] L-RSIM: A Simulation Environment for I/O Intensive Workloads. L. Schaelicke. In *Proceedings of the 3rd Annual IEEE Workshop on Workload Characterization*, pp. 83-89, 2000.

[12] RSIM: An Execution-Driven Simulator for ILP-Based Shared Memory Multiprocessors and Uniprocessors. V. Pai, P. Ranganathan, and S. Adve. In *IEEE TCCA Newsletter*, October 1997.

[13] The Interaction of Architecture and Operating System Design. T. Anderson, H. Levy, B. Bershad, and E. Lazowska. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 109-120, April 1991.

[14] Examination of a Novel Method of Emulating System Calls in Microprocessor Simulators. E. Bauer. B.S. Thesis, School of Engineering and Applied Science, University of Virginia, April 2002.

[15] Accurately Modeling Speculative Instruction Fetching in Trace-Driven Simulation. R. Bhargava, L. John, and F.Matus. In *International Performance, Computing, and Communications Conference*, pp65-71, February 1999.

[16] The Linux Kernel Archives, http://www.kernel.org/

[17] The Stanford Flash Multiprocessor. J. Kushin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. In *Proceedings of the 21st International Symposium on Computer Architecture*, pp 302-313, April 1994.

[18] Alpha Architecture Handbook, Version 4. Compaq Computer Corp., 1998.

[19] PowerPC Architecture. IBM Corp., Austin, Texas, May 1993.

[20] IA-32 Intel Architecture Software Developer's Manual, Volume 1. Intel Corp., Denver, Colorado, 2004.

[21] The SPARC Architecture Manual, Version 8. SPARC International Inc., Menlo Park, California, 1992.

[22] MIPS RISC Architecture. G. Kane and J Heinrich. Prentice-Hall, September, 1991.

[23] The Design and Implementation of an Extendible Instruction Set Simulator. P. Zadarnowski. B.S. Thesis, School of Computer Science and Engineering, University of New South Wales, November 2000.

[24] The SImulator for Multithreaded Computer Architecture, Release 1.2. J. Huang. Technical Report No: ARCTiC-00-05, Laboratory for Advanced Research in Computing Technology and Compilers, University of Minnesota, June 2000.

[25] The SimCore/Alpha Functional Simulator. K. Kise, T. Katagiri, H. Honda, and T. Yuba. In *Proceedings of the Workshop on Computer Architecture Education (WCAE-2004)*, June 2004, Munich, Germany.

[26] An Efficient Strategy for Developing a Simulator for a Novel Concurrent Multithreaded Processor Architecture. J. Huang and D. J. Lilja. In *The Sixth Int'l Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems,* July, 1998.

[27] The BOCHS User Manual. K. Lawton, B. Denny, N. Guarneri, V. Ruppert, C. Bothamy, and M. Calabrese. http://bochs.sourceforge.net/

[28] Executable and Linkable Format (ELF). Tool Interface Standards, Portable Formats Specification, Version 1.1.